

Guide for Module Developers

Everything You Need to Know about EnergyPlus Calculational Development

(but were hesitant to ask)

Date: August 2, 2004

TABLE OF CONTENTS

Introduction	1
Modules in EnergyPlus	2
What is a module anyway?	2
Program Modules	2
Data Only Modules	2
What is a module developer?	2
Input Concepts	3
Input Data Dictionary	3
Input Data File	5
Input Considerations	6
Advanced Input Considerations	8
Module Structure	10
Module Outline	10
Module Example	12
How it fits together	31
Top Level Calling Tree	31
High Level HVAC Calling Tree (schematic – not all routines are shown)	31
Air System Calling Tree (schematic – not all routines are shown)	32
Plant Supply Calling Tree (schematic – not all routines are shown)	32
Zone Equipment Calling Tree (schematic – not all routines are shown)	33
Inserting the New Module into the Program	34
Considerations for Legacy Codes	36
Code Readability vs. Speed of Execution	37
Speed of Execution	37
EnergyPlus Services	39
Global Data	39

TABLE OF CONTENTS

Utility Routines/Functions.....	39
Table 1. Table of Utility Functions	39
Input Services	40
InputProcessor	40
GetNumObjectsFound	41
GetObjectItem	41
GetObjectDefMaxArgs	42
Extensible input techniques.....	42
GetObjectItemNum	42
FindItemInList.....	42
SameString	43
VerifyName	43
RangeCheck	43
MakeUPPERCase.....	44
NodeInputManager	44
GetOnlySingleNode	45
GetNodeNums	45
Unique Node Checking	46
InitUniqueNodeCheck	46
CheckUniqueNodes	46
EndUniqueNodeCheck.....	47
SetUpCompSets and TestCompSet	47
SetUpCompSets.....	48
TestCompSet.....	49
Schedule Services	50
GetScheduleIndex.....	52

TABLE OF CONTENTS

CheckScheduleValueMinMax	52
GetScheduleMinValue	52
GetScheduleMaxValue	53
GetCurrentScheduleValue	53
Other Useful Utilities	53
GetNewUnitNumber	53
FindUnitNumber	53
FindNumberinList	54
ValidateComponent.....	54
CheckComponent	54
CreateSysTimeIntervalString	55
TrimSigDigits.....	55
RoundSigDigits	56
Error Messages.....	56
Display Strings	58
Performance Curve Services	58
GetCurveIndex	58
CurveValue	59
Fluid Property Services	59
Using Fluid Property Routines in EnergyPlus Modules.....	59
Fluid Properties Functions for Refrigerant Class Fluids.....	59
Reference Data Set (RDS) Values for Refrigerant Class Fluids	60
Table 1. Regions for Fluid Properties	60
Fluid Property Data and Expanding the Refrigerants Available to EnergyPlus	61
Fluid Properties Functions for Glycol Class Fluids.....	64
Default Values for Glycol Class Fluids	64

TABLE OF CONTENTS

Fluid Property Data and Expanding the Glycols Available to EnergyPlus64

Weather Services.....	67
Flags and Parameters.....	67
Parameters.....	67
Simulation Flags.....	67
Psychrometric services	69
PsyRhoAirFnPbTdbW (Pb,Tdb,W).....	69
PsyCpAirFnWTdb (W,Tdb)	69
PsyHfgAirFnWTdb (W,Tdb)	70
PsyHgAirFnWTdb (W,Tdb)	70
PsyTdpFnTdbTwbPb (Tdb,Twb,Pb).....	70
PsyTdpFnWPb (W,Pb).....	70
PsyHFnTdbW (Tdb,W).....	70
PsyHFnTdbRhPb (Tdb,Rh,Pb).....	70
PsyTdbFnHW (H,W)	70
PsyRhovFnTdbRh (Tdb,Rh).....	70
PsyRhovFnTdbWP (Tdb,W,Pb)	70
PsyRhFnTdbRhov (Tdb,Rhov).....	70
PsyRhFnTdbWPb (Tdb,W,Pb)	71
PsyTwbFnTdbWPb (Tdb,W,Pb).....	71
PsyVFnTdbWPb (Tdb,W,Pb)	71
PsyWFnTdpPb (Tdp,Pb)	71
PsyWFnTdbH (Tdb,H).....	71
PsyWFnTdbTwbPb (Tdb,Twb,Pb).....	71
PsyWFnTdbRhPb (Tdb,Rh,Pb)	71
PsyPsatFnTemp (T)	71

TABLE OF CONTENTS

PsyTsatFnHPb (H,Pb).....	71
PsyTsatFnPb (P).....	71
CPCW (Temp).....	72
CPHW (Temp).....	72
CVHW (Temp).....	72
RhoH2O (Temp).....	72
Tabular Output Utilities.....	72
WriteReportHeaders(reportName,objectName,averageOrSum).....	72
WriteSubtitle(subtitle)	72
WriteTable(body,rowLabels,columnLabels,widthColumn)	72
HVAC Network.....	74
Branches, Connectors, and Nodes	74
Figure 1. HVAC Input Diagram.....	74
Nodes in the simulation.....	76
Getting Nodes	77
Data Flow in an HVAC Component Module.....	77
Node Mass Flow Variables	80
Output	82
How Do I Output My Variables?	82
Table 3. SetupOutputVariable Arguments.....	83
Output Variable Dos and Don'ts.....	84
What Variables Should I Output?	84
Output Variable Naming Conventions.....	84
What are Meters?.....	84
How Do I Create A Meter?	85
Rules for Meter Variables.....	85

TABLE OF CONTENTS

Important Rules for Module Developers.....	87
Appendix A. DataGlobals Module	88
Appendix B. DataEnvironment Module	92
Appendix C. Submissions and Check-ins	95
Appendix D. Documentation Specifics	98
Appendix E. Module Template	99
Appendix F. Test File Documentation	106

Introduction

EnergyPlus is a modular simulation program designed to model the performance, energy consumption and pollutant production of a building. EnergyPlus models energy transport through the building envelope, heat gains within the building, and all the HVAC equipment used to heat and cool the building. The program is designed for ease of development. The concept is that many people will contribute to EnergyPlus and the program structure has been designed to make this possible.

EnergyPlus is written entirely in Fortran 90 with updates to Fortran 95 – all of EnergyPlus code should be at minimum Fortran 90 compliant and can accept the newer features of Fortran 95 as well. Fortran 90/95 is a powerful modern programming language with many features. Using Fortran 90/95 it is possible to program in many different styles. The EnergyPlus team has chosen a particular style that emphasizes code extensibility (ease of development), understandability, maintainability, and robustness. Less emphasis was placed on program speed and size. Fortran 90/95 has all the features that permit the creation of readable, maintainable, and extensible code. In particular, the ability to create data and program modules with various levels of data hiding allows EnergyPlus to be built out of semi-independent modules. This allows a new EnergyPlus developer to concentrate on programming a single component without having to learn the entire program and data structure.

The EnergyPlus programming style is described in the *EnergyPlus Programming Standard*. The *Programming Standard* should be consulted for details such as variable and subroutine naming conventions. In this document, we will describe the steps a developer must follow to create a new EnergyPlus component model. In particular, we will assume the developer wishes to simulate an HVAC component that cannot yet be modeled by EnergyPlus.

Modules in EnergyPlus

What is a module anyway?

Program Modules

A module is a Fortran 90/95 programming construct that can be used in various ways. In EnergyPlus, its primary use is to segment a rather large program into smaller, more manageable pieces. Each module is a separate package of source code stored on a separate file. The entire collection of modules, when compiled and linked, forms the executable code of EnergyPlus.

Each module contains source code for closely related data structures and procedures. For instance, the WeatherManager module contains all the weather handling routines in EnergyPlus. The module is contained in the file WeatherManager.f90. Another example is PlantPumps. This module contains all the code to simulate pumps in EnergyPlus. It is contained in file PlantPumps.f90.

Of course dividing a program into modules can be done in various ways. We have attempted to create modules that are as self-contained as possible. The philosophy that has been used in creating EnergyPlus is contained in the [Programming Standard](#) reference document. Logically, the modules in EnergyPlus form an inverted tree structure. At the top is EnergyPlus. Just below that are ProcessInput and ManageSimulation. At the bottom are the modules such as HVACDamperComponent that model the actual HVAC components.

Data Only Modules

EnergyPlus also uses modules that contain only data. These modules form one of the primary ways data is structured and shared in EnergyPlus. An example is the DataEnvironment module. Many parts of the program need access to the outdoor conditions. All of that data is encapsulated in DataEnvironment. Modules that need this data obtain access through a Fortran USE statement. Without such access, modules cannot use or change this data.

What is a module developer?

A module developer is someone who is going to add to the simulation capabilities of EnergyPlus. Someone, for instance, who is interested in adding code to model a new type of HVAC equipment. The most straightforward way of doing this is to create a new program module – hence the term “module developer”. Another kind of module developer would be the adaptation of an existing “legacy” code to EnergyPlus.

In EnergyPlus, the first step in creating a new component model is to define the input. So, before we discuss modules in more detail, we must first describe the EnergyPlus input.

Input Concepts

In EnergyPlus, input and output are accomplished by means of ASCII (text) files. On the input side, there are two files:

- 1) the Input Data Dictionary (IDD) that describes the types (classes) of input objects and the data associated with each object;
- 2) the Input Data File (IDF) that contains all the data for a particular simulation.

Each EnergyPlus module is responsible for getting its own input. Of course, EnergyPlus provides services to the module that make this quite easy. The first task of a module developer is to design and insert a new entry into the Input Data Dictionary.

Input Data Dictionary

An entry in the IDD consists of comma-separated text terminated by a semicolon. For instance:

```
COIL:Water:SimpleHeating,
  A1 , \field Coil Name
      \required-field
      \type alpha
      \reference HeatingCoilName
  A2 , \field Available Schedule
      \type object-list
      \object-list ScheduleNames
  N1 , \field UA of the Coil
      \units W/K
      \autosizable
  N2 , \field Max Water Flow Rate of Coil
      \units m3/s
      \autosizable
      \ip-units gal/min
  A3 , \field Coil_Water_Inlet_Node
      \required-field
  A4 , \field Coil_Water_Outlet_Node
      \required-field
  A5 , \field Coil_Air_Inlet_Node
      \required-field
  A6 ; \field Coil_Air_Outlet_Node
      \required-field
```

This entry defines a simple water-heating coil and specifies all of the input data needed to model it. The following rules apply.

- 1) The first element *COIL:Water:Simple Heating* is the class name (also called a keyword or key). This class name must be unique in the IDD. The maximum length for the class name is 60 characters. Embedded spaces are allowed and are significant.
- 2) In most cases, one should have fields following the object name. An object name by itself (terminated with a semi-colon) is a “section” – there may be uses for sections in input but the “Getting” of input is not hierarchical – one typically gets all objects of one type and then all objects of the next type.

- 3) In most cases, the second field of an object should be an "alpha" and the field name should contain the word "name". (This will allow for certain validations later on.)
- 4) Commas separate fields. They always act as separators – thus there is no way to include a comma in a class name or as part of a data field.
- 5) Similarly, semicolons are terminators – a semicolon is always interpreted as the end of an EnergyPlus "sentence". So, avoid embedded semicolons in class names or data fields.
- 6) Blank lines are allowed.
- 7) Each line can be up to 500 characters in length.
- 8) The comment character is an exclamation or a backslash. Anything on a line after an "!" or a "\" is ignored during EnergyPlus input.

The only significant syntax elements are the commas, the semi colon, the N's (denoting numeric data), and the A's (denoting alphanumeric data) and the exclamation and backslash. Everything else including blanks, end of lines, or even text that is not a comma, semicolon, N, or A is ignored. There are several style conventions in use however.

- 1) Sequence numbers are appended to the letters A or N denoting each data element. Thus, A2 is the second alphanumeric data item and N3 is the third numeric data item.
- 2) The class name contains a naming convention: *type:subtype:subsubtype*.
- 3) Backslashes denote specially formatted comments. These comments provide information about the input, such as a description of the item, units, limits, mins & maxes, etc., in a form that can be processed by an input editor or interface. A complete description of the backslash comment format is given at the start of the IDD file and in the *Guide for Interface Developers*. While these are "comments", they are quite important and allow the InputProcessor module to do some error checking for you. They are also used by the IDFEditor which many of the users continue to use (in light of so few Interfaces).
 - \default – the number (N fields) or phrase (A fields) after this special field will be filled for any input file that has a blank in that field.
 - \minimum or \minimum> -- the number following this special field will be automatically checked during input
 - \maximum or \maximum< -- the number following this special field will be automatically checked during input
 - \extensible – allows you to structure your GetInput routine so that the object arguments can be expanded (you will need notes or a memo to instruct how this is to be done)

Overall, the IDD file has very little structure. Generally, a new entry should be placed next to entries describing similar components. *COIL:Water:Simple Heating*, for instance, is grouped with entries describing other water coils.

Summary

The first task for a module developer is to create a new entry in the Input Data Dictionary. This entry defines the data needed to model the new component.

Input Data File

The Input Data File (IDF) is the file containing the data for an actual simulation. This file is also a text (ASCII) file with a syntax “filling in the blanks” of the definitions in the IDD. A portion of an IDF with input data for the hot water coil defined in the IDD example looks like:

```
!Demand Heating Components
COIL:Water:SimpleHeating,
  Reheat Coil Zone 1,      !Name of cooling coil
  FanAndCoilAvailSched,   !Cooling Coil Schedule
  400.0,                  !UA of the Coil
  1.3,                    !Max Water Flow Rate of Coil kg/sec
  Zone 1 Reheat Water Inlet Node, !Water side inlet node
  Zone 1 Reheat Water Outlet Node, !Water side outlet node
  Zone 1 Reheat Air Inlet Node,   !Air side inlet node
  Zone 1 Reheat Air Outlet Node;  !Air side outlet node

COIL:Water:SimpleHeating,
  Reheat Coil Zone 2,      !Name of cooling coil
  FanAndCoilAvailSched,   !Cooling Coil Schedule
  400.0,                  !UA of the Coil
  1.2,                    !Max Water Flow Rate of Coil kg/sec
  Zone 2 Reheat Water Inlet Node, !Water side inlet node
  Zone 2 Reheat Water Outlet Node, !Water side outlet node
  Zone 2 Reheat Air Inlet Node,   !Air side inlet node
  Zone 2 Reheat Air Outlet Node;  !Air side outlet node

COIL:Water:SimpleHeating,
  Reheat Coil Zone 3,      !Name of cooling coil
  FanAndCoilAvailSched,   !Cooling Coil Schedule
  400.0,                  !UA of the Coil
  1.8,                    !Max Water Flow Rate of Coil kg/sec
  Zone 3 Reheat Water Inlet Node, !Water side inlet node
  Zone 3 Reheat Water Outlet Node, !Water side outlet node
  Zone 3 Reheat Air Inlet Node,   !Air side inlet node
  Zone 3 Reheat Air Outlet Node;  !Air side outlet node
```

Each coil entry begins with the class name (keyword) specifying the type of coil. Next is the coil name – a user (or interface) created name that is unique within the given class. Generally in EnergyPlus, objects within a class are distinguished by unique names. The object name is usually the first data element following the class name. Any alphanumeric data item in the IDF can be up to 60 characters long. Any characters past 60 are truncated (lost). After the object name comes the real data. If we look at the IDD we see that the first data item after the object name is expected to be an alphanumeric – a schedule name. In the IDF, we see the corresponding field is “FanAndCoilAvailSched”, the object name of a schedule elsewhere in the IDF file. In EnergyPlus, all references to other data entries (objects) are via object names. The next two data items are numeric: the coil UA and the maximum water mass flow rate. The final four items are again alphanumeric – the names of the coil inlet and outlet nodes. Nodes are used in EnergyPlus to connect HVAC components together into HVAC systems.

The example illustrates the use of comments to create clear input. The IDF is intended to be human readable, largely for development and debugging purposes. Of course, most users will never see an IDF – they will interact with EnergyPlus through a Graphical

User Interface (GUI), which will write the IDF for them. However, a module developer is a special kind of user. The module developer will need to create a portion of an IDF by hand very early in the development process in order to begin testing the module under development. Thus, it is important to understand the IDF syntax and to use comments to create readable test IDF files.

Summary

One of the early tasks of a module developer is to create input (most likely by hand) for the new component and to insert it into an existing IDF file in order to test the new component model. The IDF syntax resembles the syntax for the IDD. The data follows the IDD class description. Comments should be used to make the IDF readable.

Input Considerations

The IDD/IDF concept allows the module developer much flexibility. Along with this flexibility comes a responsibility to the overall development of EnergyPlus. Developers must take care not to obstruct other developers with their additions or changes. Major changes in the IDD require collaboration among the developers (both module and interface).

In many cases, the developer may be creating a new model – a new HVAC component, for instance. Then the most straightforward approach is to create a new object class in the IDD with its own unique, self-contained input. This will seldom impact other developers.

In some cases, the developer may be adding a calculation within an existing module or for an existing class of objects. This calculation may require new or different input fields. Then the developer has a number of choices. This section will present some ideas for adding to the IDD that will minimize impact to other developers.

For example, consider the implementation of Other Side Coefficients (OSC) in the IDD. Other side coefficients are a simplification for the surface heat balance and were used mostly in BLAST 2.0 before we had interzone surfaces. We have carried this forward into EnergyPlus for those users that understand and can use it. We'll use it as an example of approaches to adding data items to the IDD. Moreover, we'll try to give some hints on which approaches might be used for future additions.

So, you're adding something to EnergyPlus and it is part of an existing module or object class. What do you do with your required inputs to your model? There are at least four options:

Embed your values in a current object class definition.

Put something in the current definition that will trigger a "GetInput" for your values.

Put something in the current definition that will signal a "special" case and embed a name (of your item) in the definition (this adds 1 or 2 properties to the object).

Just get your input and have each of those inputs reference a named object.

For example, using the OSC option in surfaces, in the beta 2 version of EnergyPlus we had

```
A8 , \field Exterior environment
    \type alpha
    \note <for Interzone Surface:Adjacent surface name>
    \note For non-interzone surfaces enter:
    \note ExteriorEnvironment, Ground, or OtherSideCoeff
    \note OSC won't use CTFs

N24, \field User selected Constant Temperature
N25, \field Coefficient modifying the user selected constant
      temperature
N26, \field Coefficient modifying the external dry bulb temperature
N27, \field Coefficient modifying the ground temperature
N28, \field Combined convective/radiative film coefficient
    \note if=0, use other coefficients
N29, \field Coefficient modifying the wind speed term (s/m)
N30, \field Coefficient modifying the zone air temperature part of
      the equation
```

1) We have done option 1: embed the values in the input. (We have also embedded these values in each and every surface derived type (internal data structure) but that can be discussed elsewhere).

When to use: It makes sense to embed these values when each and every object (SURFACE) needs these values (e.g. we need to specify Vertices for Every Surface -- so these clearly should be embedded).

After beta 2, the definition of Surfaces was changed. Obviously option 1 was not a good choice for the OSC data: the data would be rarely used. Our other options were:

2) Obviously the ExteriorEnvironment field will remain (but its name was changed to Outside Face Environment).

However, we do not want to embed the values for OtherSideCoef in the Surface items. So, if the ExteriorEnvironment continues to reference OtherSideCoef, we can easily trigger a "GetInput" for them. An additional object class would be necessary for this case.

```
OtherSideCoef, A1, \field name of OtherSideCoef,
A2, \field SurfaceName (reference to surface using OSC)
....
```

When to use: This option can be used for many cases. The same object definition will work for option 4 below. Obviously, if there is not a convenient trigger in SURFACE but you want to add a feature, this would let you do it without embedding it in the Surface Definition. If there is a trigger, such as exists with the ExteriorEnvironment, the A2 field might not be needed. This approach would become a bit cumbersome if you expected there to be a lot of these or if there were a one-to-many relationship (i.e. a single set of OSCs could be used for many surfaces). Nevertheless, the approach provides a convenient "data check"/cross reference that can be validated inside the code.

3) We could also have the SURFACE definition reference an OSC name (in this instance).

So, we'd add a field to the Surface that would be the name in the OtherSideCoef object above. Then, the OtherSideCoef objects wouldn't need a Surface Name. This is the most straightforward approach: including data in one object by referencing another, and it was the approach chosen for the redefined Surface class.

When to use: when there is a set of parameters that would be used extensively, then this would provide a name for those. If hand editing, then you only would need to change one set of these parameters rather than having to go through many. Of course, the OtherSideCoef object wouldn't also have to have the true numbers but could reference yet a third named object..... (starting to get messy).

4) We could have the OtherSideCoef object as above and just "get" it as a matter of course. (e.g., In the case where we don't have a convenient trigger such as ExteriorEnvironment).

When to use: Note that the same structure for 2 works here too. It's just not triggered (to get the input) by a value in the other object (SURFACE).

Summary

There are several approaches to adding items to the IDD. Developers need to consider impacts to other developers and users early in the implementation planning.

Advanced Input Considerations

Creating a new module/adding a new feature to EnergyPlus is a good accomplishment. However, it is likely that future additions will be done and will impact any objects created. In this regard, we ask that module developers take a longer view than "just getting my thing" going.

For example, in the "Fan Coil" object, prior to the V1.2 release, the object definition specified a cooling coil name. *But it did not specify a cooling coil type.* Rather than restrict coil names to be unique over all coils (which becomes difficult as more coil types are added), the developers only have unique names within a type. Thus, it would become difficult for the Fan Coil module to get the proper link to the correct cooling coil.

In the V1.2 release, a cooling coil type was added to the object. But in a less readable way than could have been done from the beginning. Namely, the coil type is at the end of the object whereas the coil name is in the middle of the object definition.

The "standard" for describing such fields would be to list the "coil type" and then the "coil name" fields, such as in the UNITARYSYSTEM:HEATPUMP:AIRTOAIR object.

The point is – try to envision future changes in making up objects, even if you think "that will never happen". It does not have to try to address every future case, only the most likely.


```

FAN COIL UNIT:4 PIPE,
    \min-fields 21
A1 , \field name of fan coil unit
    \required-field
A2 , \field availability schedule
    \required-field
    \type object-list
    \object-list ScheduleNames
N1 , \field maximum air flow rate
    \required-field
    \autosizable
    \units m3/s
N2 , \field maximum outside air flow rate
    \required-field
    \autosizable
    \units m3/s
A3 , \field air inlet node
    \required-field
<snip>
A11, \field cooling coil name
    \required-field
N3 , \field maximum cold water flow
    \required-field
    \autosizable
    \units m3/s
    \ip-units gal/min
<snip>
A13; \field Cooling coil type
    \required-field
    \type choice
    \key COIL:Water:SimpleCooling
    \key COIL:Water:DetailedFlatCooling
    \key COIL:Water:CoolingHeatExchangerAssisted

```

Module Structure

Let us assume that the novice EnergyPlus developer wishes to model a new HVAC component called *NewHVACComponent*. Right at the start there is a choice to make: whether to insert the new model into an existing module or to create an entirely new EnergyPlus component simulation model. Creating a new module is the easier option to explain, implement and test. We will discuss this option in this document. The discussion should also impart enough information to allow a new developer to insert a model into an existing EnergyPlus module if that option is chosen.

Module Outline

The developer will create a new file *NewHVACComponent.f90*. The file shall contain the following elements:

Note – even if your component does not need some of the suggested modules, you should include “stub” routines for these.

MODULE *NewHVACComponent*

Documentation: Fortran comments describing and documenting the module. Included are sections showing module author, module creation date, date modified and modification author. Each routine and/or function should also follow the documentation guidelines as shown in the templates.

USE Statements: Fortran statements naming other modules that this module can access, either for data or for routines.

Module Parameters: If you will be implementing more than one “type” of component in the module, it is a good idea to assign numeric parameters to each type so as to retain readability yet reduce alpha comparisons which are notoriously slow for most systems. Assign numeric parameters to alphanumeric fields within a class type (.e.g. object UnitarySystem:HeatPump, field Fan Placement: “blow through” or “draw through”) when this information is required in init, calc, update or report subroutines to further reduce alpha comparisons. Use string comparison only in GetInput subroutines.

Module Datastructure Definitions: Using the Fortran TYPE statement define the data structures needed in the module that will not be available from other modules. Define all module level variables that will be needed.

CONTAINS

SUBROUTINE *SimNewHVACComponent*

This routine selects the individual component being simulated and calls the other module subroutines that do the real work. This routine is the only routine in the module that is accessible outside the module (*PUBLIC*). All other routines in the module are *PRIVATE* and are only callable within the module. This routine is sometimes called the “driver” routine for the module.

END SUBROUTINE *SimNewHVACComponent*

SUBROUTINE *GetNewHVACComponentInput*

This routine uses the “get” routines from the InputProcessor module to obtain input for NewHVACComponent. The module data arrays are allocated and the data is moved into the arrays.

END SUBROUTINE *GetNewHVACComponentInput*

SUBROUTINE *InitNewHVACComponent*

This routine performs whatever initialization calculations that may be needed at various points in the simulation. For instance, some calculations may only need to be done once; some may need to be done at the start of each simulation weather period; some at the start of each HVAC simulation time step; and some at the start of each loop solution. This routine also transfers data from the component inlet nodes to the component data arrays every time the component is simulated, in preparation for the actual component simulation.

END SUBROUTINE *InitNewHVACComponent*

SUBROUTINE *SizeNewHVACComponent*

This routine can create the sizing options (if applicable) for the component or be left as a placeholder for later manipulation for sizing purposes.

END SUBROUTINE *SizeNewHVACComponent*

SUBROUTINE *CalcNewHVACComponent*

This routine does the actual calculations to simulate the performance of the component. Only calculation is done – there is no moving of data from or to input or output areas. There may be more than one “CALC” subroutine if more than one component is being modeled within this module.

END SUBROUTINE *CalcNewHVACComponent*

SUBROUTINE *UpdateNewHVACComponent*

This routine moves the results of the “Calc” routine(s) to the component outlet nodes.

END SUBROUTINE *UpdateNewHVACComponent*

SUBROUTINE *ReportNewHVACComponent*

This routine performs any special calculations that are needed purely for reporting purposes.

END SUBROUTINE *ReportNewHVACComponent*

END MODULE *NewHVACComponent*

Module Example

Module Fans

```

! Module containing the fan simulation routines

! MODULE INFORMATION:
!   AUTHOR      Richard J. Liesen
!   DATE WRITTEN April 1998
!   MODIFIED     Shirey, May 2001
!   RE-ENGINEERED na

! PURPOSE OF THIS MODULE:
! To encapsulate the data and algorithms required to
! manage the Fan System Component

! REFERENCES: none

! OTHER NOTES: none

! USE STATEMENTS:
! Use statements for data only modules
USE DataLoopNode
USE DataHVACGlobals, ONLY: TurnFansOn, TurnFansOff, Main, Cooling, Heating, Other, &
    OnOffFanPartLoadFraction, SmallAirVolFlow
USE DataGlobals, ONLY: SetupOutputVariable, BeginEnvrnFlag, BeginDayFlag, MaxNameLength, &
    ShowWarningError, ShowFatalError, ShowSevereError, HourOfDay, &
    SysSizingCalc, CurrentTime, OutputFileDebug, ShowContinueError
USE DataEnvironment, ONLY: StdBaroPress, DayOfMonth, Month
USE Psychrometrics, ONLY: PsyRhoAirFnPbTdbW, PsyTdbFnHW, PsyCpAirFnWTdb

! Use statements for access to subroutines in other modules
USE ScheduleManager

IMPLICIT NONE          ! Enforce explicit typing of all variables

PRIVATE ! Everything private unless explicitly made public

!MODULE PARAMETER DEFINITIONS
INTEGER, PARAMETER :: FanType_SimpleConstVolume = 1
INTEGER, PARAMETER :: FanType_SimpleVAV         = 2
INTEGER, PARAMETER :: FanType_SimpleOnOff        = 3
INTEGER, PARAMETER :: FanType_ZoneExhaust       = 4

! DERIVED TYPE DEFINITIONS
TYPE FanEquipConditions
  CHARACTER(len=MaxNameLength) :: FanName ! Name of the fan
  CHARACTER(len=MaxNameLength) :: FanType ! Type of Fan ie. Simple, Vane axial, Centrifugal, etc.
  CHARACTER(len=MaxNameLength) :: Schedule ! Fan Operation Schedule
  CHARACTER(len=MaxNameLength) :: Control ! ie. Const Vol, Variable Vol
  Integer :: SchedPtr ! Pointer to the correct schedule
  REAL :: InletAirMassFlowRate !MassFlow through the Fan being Simulated [kg/Sec]
  REAL :: OutletAirMassFlowRate
  Real :: MaxAirFlowRate !Max Specified Volume Flow Rate of Fan [m^3/sec]
  Real :: MinAirFlowRate !Min Specified Volume Flow Rate of Fan [m^3/sec]
  REAL :: MaxAirMassFlowRate ! Max flow rate of fan in kg/sec
  REAL :: MinAirMassFlowRate ! Min flow rate of fan in kg/sec
  REAL :: InletAirTemp
  REAL :: OutletAirTemp
  REAL :: InletAirHumRat
  REAL :: OutletAirHumRat
  REAL :: InletAirEnthalpy
  REAL :: OutletAirEnthalpy
  REAL :: FanPower !Power of the Fan being Simulated [kW]
  REAL :: FanEnergy !Fan energy in [kJ]
  REAL :: FanRuntimeFraction !Fraction of the timestep that the fan operates
  REAL :: DeltaTemp !Temp Rise across the Fan [C]
  REAL :: DeltaPress !Delta Pressure Across the Fan [N/M^2]
  REAL :: FanEff !Fan total efficiency; motor and mechanical
  REAL :: MotEff !Fan motor efficiency
  REAL :: MotInAirFrac !Fraction of motor heat entering air stream

```

```

REAL, Dimension(5):: FanCoeff      !Fan Part Load Coefficients to match fan type
! Mass Flow Rate Control Variables
REAL      :: MassFlowRateMaxAvail
REAL      :: MassFlowRateMinAvail
REAL      :: RhoAirStdInit
INTEGER   :: InletNodeNum
INTEGER   :: OutletNodeNum
END TYPE FanEquipConditions

!MODULE VARIABLE DECLARATIONS:
INTEGER :: NumFans      ! The Number of Fans found in the Input
TYPE (FanEquipConditions), ALLOCATABLE, DIMENSION(:) :: Fan

! Subroutine Specifications for the Module
! Driver/Manager Routines
Public SimulateFanComponents

! Get Input routines for module
PRIVATE GetFanInput

! Initialization routines for module
PRIVATE InitFan
PRIVATE SizeFan

! Algorithms for the module
Private SimSimpleFan
PRIVATE SimVariableVolumeFan
PRIVATE SimZoneExhaustFan

! Update routine to check convergence and update nodes
Private UpdateFan

! Reporting routines for module
Private ReportFan

CONTAINS

! MODULE SUBROUTINES:
!*****
SUBROUTINE SimulateFanComponents(CompName,FirstHVACIteration)

! SUBROUTINE INFORMATION:
!   AUTHOR      Richard Liesen
!   DATE WRITTEN February 1998
!   MODIFIED    na
!   RE-ENGINEERED na

! PURPOSE OF THIS SUBROUTINE:
! This subroutine manages Fan component simulation.

! METHODOLOGY EMPLOYED:
! na

! REFERENCES:
! na

! USE STATEMENTS:
USE InputProcessor, ONLY: FindItemInList

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

! SUBROUTINE ARGUMENT DEFINITIONS:
CHARACTER(len=*) , INTENT(IN) :: CompName
LOGICAL,          INTENT (IN):: FirstHVACIteration

! SUBROUTINE PARAMETER DEFINITIONS:
! na

! INTERFACE BLOCK SPECIFICATIONS

```

```

        ! DERIVED TYPE DEFINITIONS
        ! na

        ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
INTEGER      :: FanNum      ! current fan number
LOGICAL,SAVE :: GetInputFlag = .True. ! Flag set to make sure you get input once

        ! FLOW:

! Obtains and Allocates fan related parameters from input file
IF (GetInputFlag) THEN !First time subroutine has been entered
    CALL GetFanInput
    GetInputFlag=.false.
End If

! Find the correct FanNumber with the AirLoop & CompNum from AirLoop Derived Type
!FanNum = AirLoopEquip(AirLoopNum)%ComponentOfTypeNum(CompNum)
! Determine which Fan given the Fan Name
FanNum = FindItemInList(CompName,Fan%FanName,NumFans)
IF (FanNum == 0) THEN
    CALL ShowFatalError('Fan not found='//TRIM(CompName))
ENDIF

! With the correct FanNum Initialize
CALL InitFan(FanNum,FirstHVACIteration) ! Initialize all fan related parameters

! Calculate the Correct Fan Model with the current FanNum
IF (Fan(FanNum)%FanType_Num == FanType_SimpleConstVolume) THEN
    Call SimSimpleFan(FanNum)
Else IF (Fan(FanNum)%FanType_Num == FanType_SimpleVAV) THEN
    Call SimVariableVolumeFan(FanNum)
Else If (Fan(FanNum)%FanType_Num == FanType_SimpleOnOff) THEN
    Call SimOnOffFan(FanNum)
Else If (Fan(FanNum)%FanType_Num == FanType_ZoneExhaust) THEN
    Call SimZoneExhaustFan(FanNum)
End If
! Update the current fan to the outlet nodes
Call UpdateFan(FanNum)

! Report the current fan
Call ReportFan(FanNum)

RETURN

END SUBROUTINE SimulateFanComponents

! Get Input Section of the Module
!*****
SUBROUTINE GetFanInput

        ! SUBROUTINE INFORMATION:
        !      AUTHOR      Richard Liesen
        !      DATE WRITTEN April 1998
        !      MODIFIED     Shirey, May 2001
        !      RE-ENGINEERED na

        ! PURPOSE OF THIS SUBROUTINE:
        ! Obtains input data for fans and stores it in fan data structures

        ! METHODOLOGY EMPLOYED:
        ! Uses "Get" routines to read in data.

        ! REFERENCES:
        ! na

        ! USE STATEMENTS:
USE InputProcessor
USE NodeInputManager, ONLY: GetOnlySingleNode
USE CurveManager, ONLY: GetCurveIndex

```

```

USE BranchInputManager, ONLY: TestCompSet

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

      ! SUBROUTINE ARGUMENT DEFINITIONS:
      ! na

      ! SUBROUTINE PARAMETER DEFINITIONS:
      ! na

      ! INTERFACE BLOCK SPECIFICATIONS
      ! na

      ! DERIVED TYPE DEFINITIONS
      ! na

      ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
INTEGER :: FanNum      ! The fan that you are currently loading input into
INTEGER :: NumSimpFan  ! The number of Simple Const Vol Fans
INTEGER :: NumVarVolFan ! The number of Simple Variable Vol Fans
INTEGER :: NumOnOff    ! The number of Simple on-off Fans
INTEGER :: NumZoneExhFan
INTEGER :: SimpFanNum
INTEGER :: OnOffFanNum
INTEGER :: VarVolFanNum
INTEGER :: ExhFanNum
INTEGER :: NumAlphas
INTEGER :: NumNums
INTEGER :: IOSTAT
REAL, DIMENSION(11) :: NumArray
CHARACTER(len=MaxNameLength), DIMENSION(4) :: AlphArray
LOGICAL :: ErrorsFound = .false.  ! If errors detected in input
LOGICAL :: IsNotOK      ! Flag to verify name
LOGICAL :: IsBlank      ! Flag for blank name

      ! Flow
NumSimpFan = GetNumObjectsFound('FAN:SIMPLE:CONSTVOLUME')
NumVarVolFan = GetNumObjectsFound('FAN:SIMPLE:VARIABLEVOLUME')
NumOnOff = GetNumObjectsFound('FAN:SIMPLE:ONOFF')
NumZoneExhFan = GetNumObjectsFound('ZONE EXHAUST FAN')
NumFans = NumSimpFan + NumVarVolFan + NumZoneExhFan + NumOnOff
IF (NumFans > 0) THEN
  ALLOCATE(Fan(NumFans))
  ! Initialize fan data structures
  Fan%FanName = ' '
  Fan%FanType = ' '
  Fan%Schedule = ' '
  Fan%Control = ' '
  Fan%SchedPtr = 0
  Fan%InletAirMassFlowRate = 0.0
  Fan%OutletAirMassFlowRate = 0.0
  Fan%MaxAirFlowRate = 0.0
  Fan%MinAirFlowRate = 0.0
  Fan%MaxAirMassFlowRate = 0.0
  Fan%MinAirMassFlowRate = 0.0
  Fan%InletAirTemp = 0.0
  Fan%OutletAirTemp = 0.0
  Fan%InletAirHumRat = 0.0
  Fan%OutletAirHumRat = 0.0
  Fan%InletAirEnthalpy = 0.0
  Fan%OutletAirEnthalpy = 0.0
  Fan%FanPower = 0.0
  Fan%FanEnergy = 0.0
  Fan%FanRuntimeFraction = 0.0
  Fan%DeltaTemp = 0.0
  Fan%DeltaPress = 0.0
  Fan%FanEff = 0.0
  Fan%MotEff = 0.0
  Fan%MotInAirFrac = 0.0
  Fan%FanCoeff(1) = 0.0
  Fan%FanCoeff(2) = 0.0

```

```

Fan%FanCoeff(3) = 0.0
Fan%FanCoeff(4) = 0.0
Fan%FanCoeff(5) = 0.0
Fan%MassFlowRateMaxAvail = 0.0
Fan%MassFlowRateMinAvail = 0.0
Fan%RhoAirStdInit = 0.0
Fan%InletNodeNum = 0
Fan%OutletNodeNum = 0
ENDIF

DO SimpFanNum = 1, NumSimpFan
  FanNum = SimpFanNum
  CALL GetObjectItem('FAN:SIMPLE:CONSTVOLUME',SimpFanNum,AlphArray, &
    NumAlphas,NumArray,NumNums,IOSTAT)

  IsNotOK=.false.
  IsBlank=.false.
  CALL VerifyName(AlphArray(1),Fan%FanName,FanNum-1,IsNotOK,IsBlank, &
    'FAN:SIMPLE:CONSTVOLUME Name')

  IF (IsNotOK) THEN
    ErrorsFound=.true.
    IF (IsBlank) AlphArray(1)='xxxxx'
  ENDIF
  Fan(FanNum)%FanName = AlphArray(1)
  Fan(FanNum)%FanType = 'SIMPLE'
  Fan(FanNum)%Schedule = AlphArray(2)
  Fan(FanNum)%SchedPtr =GetScheduleIndex(AlphArray(2))
  IF (Fan(FanNum)%SchedPtr == 0) THEN
    CALL ShowSevereError('FAN:SIMPLE:CONSTVOLUME, Schedule not found='//TRIM(AlphArray(2)))
    ErrorsFound=.true.
  ENDIF
  Fan(FanNum)%Control = 'CONSTVOLUME'

  Fan(FanNum)%FanEff      = NumArray(1)
  Fan(FanNum)%DeltaPress  = NumArray(2)
  Fan(FanNum)%MaxAirFlowRate= NumArray(3)
  Fan(FanNum)%MotEff      = NumArray(4)
  Fan(FanNum)%MotInAirFrac = NumArray(5)
  Fan(FanNum)%MinAirFlowRate= 0.0

  Fan(FanNum)%InletNodeNum = &
    GetOnlySingleNode(AlphArray(3),ErrorsFound,'Fan:Simple:ConstVolume',AlphArray(1), &
      NodeType_Air,NodeConnectionType_Inlet,1,ObjectIsNotParent)
  Fan(FanNum)%OutletNodeNum = &
    GetOnlySingleNode(AlphArray(4),ErrorsFound,'Fan:Simple:ConstVolume',AlphArray(1), &
      NodeType_Air,NodeConnectionType_Outlet,1,ObjectIsNotParent)

  CALL TestCompSet('FAN:SIMPLE:CONSTVOLUME',AlphArray(1),AlphArray(3), &
    AlphArray(4),'Air Nodes')

END DO ! end Number of Simple FAN Loop

DO VarVolFanNum = 1, NumVarVolFan
  FanNum = NumSimpFan + VarVolFanNum
  CALL GetObjectItem('FAN:SIMPLE:VARIABLEVOLUME',VarVolFanNum,AlphArray, &
    NumAlphas,NumArray,NumNums,IOSTAT)

  IsNotOK=.false.
  IsBlank=.false.
  CALL VerifyName(AlphArray(1),Fan%FanName,FanNum-1,IsNotOK,IsBlank, &
    'FAN:SIMPLE:VARIABLEVOLUME Name')

  IF (IsNotOK) THEN
    ErrorsFound=.true.
    IF (IsBlank) AlphArray(1)='xxxxx'
  ENDIF
  Fan(FanNum)%FanName = AlphArray(1)
  Fan(FanNum)%FanType = 'SIMPLE'
  Fan(FanNum)%Schedule = AlphArray(2)
  Fan(FanNum)%SchedPtr =GetScheduleIndex(AlphArray(2))
  IF (Fan(FanNum)%SchedPtr == 0) THEN
    CALL ShowSevereError('FAN:SIMPLE:VARIABLEVOLUME, Schedule not found='// &

```



```

                                TRIM(AlphaArray(2)))

    ErrorsFound=.true.
ENDIF
Fan(FanNum)%Control = 'VARIABLEVOLUME'

Fan(FanNum)%FanEff      = NumArray(1)
Fan(FanNum)%DeltaPress  = NumArray(2)
Fan(FanNum)%MaxAirFlowRate= NumArray(3)
Fan(FanNum)%MinAirFlowRate= NumArray(4)
Fan(FanNum)%MotEff      = NumArray(5)
Fan(FanNum)%MotInAirFrac = NumArray(6)
Fan(FanNum)%FanCoeff(1) = NumArray(7)
Fan(FanNum)%FanCoeff(2) = NumArray(8)
Fan(FanNum)%FanCoeff(3) = NumArray(9)
Fan(FanNum)%FanCoeff(4) = NumArray(10)
Fan(FanNum)%FanCoeff(5) = NumArray(11)
IF (Fan(FanNum)%FanCoeff(1) == 0.0 .and. Fan(FanNum)%FanCoeff(2) == 0.0 .and. &
    Fan(FanNum)%FanCoeff(3) == 0.0 .and. Fan(FanNum)%FanCoeff(4) == 0.0 .and. &
    Fan(FanNum)%FanCoeff(5) == 0.0) THEN
    CALL ShowWarningError('Fan Coefficients are all zero. No Fan power will be reported.')
    CALL ShowContinueError('For Fan:Simple:VariableVolume, Fan='//TRIM(AlphaArray(1)))
ENDIF
Fan(FanNum)%InletNodeNum = &

GetOnlySingleNode(AlphaArray(3),ErrorsFound,'Fan:Simple:VariableVolume',AlphaArray(1), &
    NodeType_Air,NodeConnectionType_Inlet,1,ObjectIsNotParent)
Fan(FanNum)%OutletNodeNum = &

GetOnlySingleNode(AlphaArray(4),ErrorsFound,'Fan:Simple:VariableVolume',AlphaArray(1), &
    NodeType_Air,NodeConnectionType_Outlet,1,ObjectIsNotParent)

    CALL TestCompSet('FAN:SIMPLE:VARIABLEVOLUME',AlphaArray(1),AlphaArray(3),AlphaArray(4), &
        'Air Nodes')

END DO      ! end Number of Variable Volume FAN Loop

DO ExhFanNum = 1, NumZoneExhFan
    FanNum = NumSimpFan + NumVarVolFan + ExhFanNum
    CALL GetObjectItem('ZONE EXHAUST FAN',ExhFanNum,AlphaArray, &
        NumAlphas,NumArray,NumNums,IOSTAT)

    IsNotOK=.false.
    IsBlank=.false.
    CALL VerifyName(AlphaArray(1),Fan%FanName,FanNum-1,IsNotOK,IsBlank,'ZONE EXHAUST FAN Name')
    IF (IsNotOK) THEN
        ErrorsFound=.true.
        IF (IsBlank) AlphaArray(1)='xxxxx'
    ENDIF
    Fan(FanNum)%FanName = AlphaArray(1)
    Fan(FanNum)%FanType = 'ZONE EXHAUST FAN'
    Fan(FanNum)%Schedule = AlphaArray(2)
    Fan(FanNum)%SchedPtr =GetScheduleIndex(AlphaArray(2))
    IF (Fan(FanNum)%SchedPtr == 0) THEN
        CALL ShowSevereError('ZONE EXHAUST FAN, Schedule not found='//TRIM(AlphaArray(2)))
        ErrorsFound=.true.
    ENDIF
    Fan(FanNum)%Control = 'CONSTVOLUME'

    Fan(FanNum)%FanEff      = NumArray(1)
    Fan(FanNum)%DeltaPress  = NumArray(2)
    Fan(FanNum)%MaxAirFlowRate= NumArray(3)
    Fan(FanNum)%MotEff      = 1.0
    Fan(FanNum)%MotInAirFrac = 1.0
    Fan(FanNum)%MinAirFlowRate= 0.0

    Fan(FanNum)%InletNodeNum = &
        GetOnlySingleNode(AlphaArray(3),ErrorsFound,'Zone Exhaust Fan',AlphaArray(1), &
            NodeType_Air,NodeConnectionType_Inlet,1,ObjectIsNotParent)
    Fan(FanNum)%OutletNodeNum = &
        GetOnlySingleNode(AlphaArray(4),ErrorsFound,'Zone Exhaust Fan',AlphaArray(1), &
            NodeType_Air,NodeConnectionType_Outlet,1,ObjectIsNotParent)

```

```

! Component sets not setup yet for zone equipment
! CALL TestCompSet('ZONE EXHAUST FAN',AlphaArray(1),AlphaArray(3),AlphaArray(4),'Air Nodes')

END DO    ! end of Zone Exhaust Fan loop

DO OnOffFanNum = 1, NumOnOff
  FanNum = NumSimpFan + NumVarVolFan + NumZoneExhFan + OnOffFanNum
  CALL GetObjectItem('FAN:SIMPLE:ONOFF',OnOffFanNum,AlphaArray, &
    NumAlphas,NumArray,NumNums,IOSTAT)

  IsNotOK=.false.
  IsBlank=.false.
  CALL VerifyName(AlphaArray(1),Fan%FanName,FanNum-1,IsNotOK,IsBlank,'FAN:SIMPLE:ONOFF Name')
  IF (IsNotOK) THEN
    ErrorsFound=.true.
    IF (IsBlank) AlphaArray(1)='xxxxx'
  ENDIF
  Fan(FanNum)%FanName = AlphaArray(1)
  Fan(FanNum)%FanType = 'SIMPLE'
  Fan(FanNum)%Schedule = AlphaArray(2)
  Fan(FanNum)%SchedPtr =GetScheduleIndex(AlphaArray(2))
  IF (Fan(FanNum)%SchedPtr == 0) THEN
    CALL ShowSevereError('FAN:SIMPLE:ONOFF, Schedule not found='//TRIM(AlphaArray(2)))
    ErrorsFound=.true.
  ENDIF
  Fan(FanNum)%Control = 'ONOFF'

  Fan(FanNum)%FanEff      = NumArray(1)
  Fan(FanNum)%DeltaPress  = NumArray(2)
  Fan(FanNum)%MaxAirFlowRate= NumArray(3)
  Fan(FanNum)%MotEff      = NumArray(4)
  Fan(FanNum)%MotInAirFrac = NumArray(5)
  Fan(FanNum)%MinAirFlowRate= 0.0

  Fan(FanNum)%InletNodeNum = &
    GetOnlySingleNode(AlphaArray(3),ErrorsFound,'Fan:Simple:OnOff',AlphaArray(1), &
      NodeType_Air,NodeConnectionType_Inlet,1,ObjectIsNotParent)
  Fan(FanNum)%OutletNodeNum = &
    GetOnlySingleNode(AlphaArray(4),ErrorsFound,'Fan:Simple:OnOff',AlphaArray(1), &
      NodeType_Air,NodeConnectionType_Outlet,1,ObjectIsNotParent)

  CALL TestCompSet('FAN:SIMPLE:ONOFF',AlphaArray(1),AlphaArray(3),AlphaArray(4),'Air Nodes')

END DO    ! end Number of Simple ON-OFF FAN Loop

IF (ErrorsFound) THEN
  CALL ShowFatalError('Errors found in getting Fan input')
ENDIF

Do FanNum=1,NumFans
  ! Setup Report variables for the Fans
  CALL SetupOutputVariable('Fan Electric Power[W]', Fan(FanNum)%FanPower, &
    'System','Average',Fan(FanNum)%FanName)
  CALL SetupOutputVariable('Fan Delta Temp[C]', Fan(FanNum)%DeltaTemp, &
    'System','Average',Fan(FanNum)%FanName)
  CALL SetupOutputVariable('Fan Electric Consumption[J]', Fan(FanNum)%FanEnergy, &
    'System','Sum',Fan(FanNum)%FanName, &
    ResourceTypeKey='Electric',EndUseKey='Fans',GroupKey='System')

END DO

DO OnOffFanNum = 1, NumOnOff
  FanNum = NumSimpFan + NumVarVolFan + NumZoneExhFan + OnOffFanNum
  CALL SetupOutputVariable('On/Off Fan Runtime Fraction', &
    Fan(FanNum)%FanRuntimeFraction, 'System','Average', &
    Fan(FanNum)%FanName)

END DO

RETURN

```

```

END SUBROUTINE GetFanInput

! End of Get Input subroutines for the HB Module
! *****

! Beginning Initialization Section of the Module
! *****

SUBROUTINE InitFan(FanNum,FirstHVACIteration)

    ! SUBROUTINE INFORMATION:
    !     AUTHOR      Richard J. Liesen
    !     DATE WRITTEN February 1998
    !     MODIFIED     na
    !     RE-ENGINEERED na

    ! PURPOSE OF THIS SUBROUTINE:
    ! This subroutine is for initializations of the Fan Components.

    ! METHODOLOGY EMPLOYED:
    ! Uses the status flags to trigger initializations.

    ! REFERENCES:
    ! na

    ! USE STATEMENTS:
    USE DataSizing, ONLY: CurSysNum
    USE DataAirLoop, ONLY: AirLoopControlInfo

    IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

    ! SUBROUTINE ARGUMENT DEFINITIONS:
    LOGICAL, INTENT (IN):: FirstHVACIteration
    Integer, Intent(IN) :: FanNum

    ! SUBROUTINE PARAMETER DEFINITIONS:
    ! na

    ! INTERFACE BLOCK SPECIFICATIONS
    ! na

    ! DERIVED TYPE DEFINITIONS
    ! na

    ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
    Integer          :: InletNode
    Integer          :: OutletNode
    Integer          :: InNode
    Integer          :: OutNode
    LOGICAL,SAVE     :: MyOneTimeFlag = .true.
    LOGICAL, ALLOCATABLE,Save, DIMENSION(:) :: MyEnvrnFlag
    LOGICAL, ALLOCATABLE,Save, DIMENSION(:) :: MySizeFlag

    ! FLOW:

    IF (MyOneTimeFlag) THEN

        ALLOCATE(MyEnvrnFlag(NumFans))
        ALLOCATE(MySizeFlag(NumFans))
        MyEnvrnFlag = .TRUE.
        MySizeFlag = .TRUE.

        MyOneTimeFlag = .false.

    END IF

    IF ( .NOT. SysSizingCalc .AND. MySizeFlag(FanNum)) THEN

```

```

CALL SizeFan(FanNum)
! Set the loop cycling flag
IF (Fan(FanNum)%Control == 'ONOFF') THEN
  IF (CurSysNum > 0) THEN
    AirLoopControlInfo(CurSysNum)%CyclingFan = .TRUE.
  END IF
END IF

MySizeFlag(FanNum) = .FALSE.
END IF

! Do the Begin Environment initializations
IF (BeginEnvrnFlag .and. MyEnvrnFlag(FanNum)) THEN

  !For all Fan inlet nodes convert the Volume flow to a mass flow
  InNode = Fan(FanNum)%InletNodeNum
  OutNode = Fan(FanNum)%OutletNodeNum
  Fan(FanNum)%RhoAirStdInit = PsyRhoAirFnPbTdbW(StdBaroPress,20.0,0.0)

  !Change the Volume Flow Rates to Mass Flow Rates

  Fan(FanNum)%MaxAirMassFlowRate = Fan(FanNum)%MaxAirFlowRate * Fan(FanNum)%RhoAirStdInit
  Fan(FanNum)%MinAirMassFlowRate = Fan(FanNum)%MinAirFlowRate * Fan(FanNum)%RhoAirStdInit

  !Init the Node Control variables
  Node(OutNode)%MassFlowRateMax = Fan(FanNum)%MaxAirMassFlowRate
  Node(OutNode)%MassFlowRateMin = Fan(FanNum)%MinAirMassFlowRate

  !Initialize all report variables to a known state at beginning of simulation
  Fan(FanNum)%FanPower = 0.0
  Fan(FanNum)%DeltaTemp = 0.0
  Fan(FanNum)%FanEnergy = 0.0

  MyEnvrnFlag(FanNum) = .FALSE.
END IF

IF (.not. BeginEnvrnFlag) THEN
  MyEnvrnFlag(FanNum) = .true.
ENDIF

! Do the Begin Day initializations
! none

! Do the begin HVAC time step initializations
! none

! Do the following initializations (every time step): This should be the info from
! the previous components outlets or the node data in this section.

! Do a check and make sure that the max and min available(control) flow is
! between the physical max and min for the Fan while operating.

InletNode = Fan(FanNum)%InletNodeNum
OutletNode = Fan(FanNum)%OutletNodeNum

Fan(FanNum)%MassFlowRateMaxAvail = MIN(Node(OutletNode)%MassFlowRateMax, &
                                         Node(InletNode)%MassFlowRateMaxAvail)
Fan(FanNum)%MassFlowRateMinAvail = MIN(MAX(Node(OutletNode)%MassFlowRateMin, &
                                         Node(InletNode)%MassFlowRateMinAvail), &
                                         Node(InletNode)%MassFlowRateMaxAvail)

! Load the node data in this section for the component simulation
!
!First need to make sure that the massflowrate is between the max and min avail.
IF (Fan(FanNum)%FanType .NE. 'ZONE EXHAUST FAN') THEN
  Fan(FanNum)%InletAirMassFlowRate = Min(Node(InletNode)%MassFlowRate, &
                                         Fan(FanNum)%MassFlowRateMaxAvail)
  Fan(FanNum)%InletAirMassFlowRate = Max(Fan(FanNum)%InletAirMassFlowRate, &

```

```

                                Fan(FanNum)%MassFlowRateMinAvail)
ELSE ! zone exhaust fans - always run at the max
  Fan(FanNum)%MassFlowRateMaxAvail = Fan(FanNum)%MaxAirMassFlowRate
  Fan(FanNum)%MassFlowRateMinAvail = 0.0
  Fan(FanNum)%InletAirMassFlowRate = Fan(FanNum)%MassFlowRateMaxAvail
END IF

!Then set the other conditions
Fan(FanNum)%InletAirTemp      = Node(InletNode)%Temp
Fan(FanNum)%InletAirHumRat    = Node(InletNode)%HumRat
Fan(FanNum)%InletAirEnthalpy  = Node(InletNode)%Enthalpy

RETURN

END SUBROUTINE InitFan

SUBROUTINE SizeFan(FanNum)

  ! SUBROUTINE INFORMATION:
  !   AUTHOR          Fred Buhl
  !   DATE WRITTEN    September 2001
  !   MODIFIED        na
  !   RE-ENGINEERED   na

  ! PURPOSE OF THIS SUBROUTINE:
  ! This subroutine is for sizing Fan Components for which flow rates have not been
  ! specified in the input.

  ! METHODOLOGY EMPLOYED:
  ! Obtains flow rates from the zone or system sizing arrays.

  ! REFERENCES:
  ! na

  ! USE STATEMENTS:
USE DataSizing

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

  ! SUBROUTINE ARGUMENT DEFINITIONS:
Integer, Intent(IN) :: FanNum

  ! SUBROUTINE PARAMETER DEFINITIONS:
  ! na

  ! INTERFACE BLOCK SPECIFICATIONS
  ! na

  ! DERIVED TYPE DEFINITIONS
  ! na

  ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
REAL :: FanMinAirFlowRate
EXTERNAL ReportSizingOutput

FanMinAirFlowRate = 0.0
IF (Fan(FanNum)%MaxAirFlowRate == AutoSize) THEN

  IF (CurSysNum > 0) THEN

    CALL CheckSysSizing('FAN:'//TRIM(Fan(FanNum)%FanType)// ':' // TRIM(Fan(FanNum)%Control), &
                       Fan(FanNum)%FanName)

    SELECT CASE(CurDuctType)
    CASE(Main)
      Fan(FanNum)%MaxAirFlowRate = FinalSysSizing(CurSysNum)%DesMainVolFlow
      FanMinAirFlowRate = CalcSysSizing(CurSysNum)%SysAirMinFlowRat *
CalcSysSizing(CurSysNum)%DesMainVolFlow
    CASE(Cooling)
      Fan(FanNum)%MaxAirFlowRate = FinalSysSizing(CurSysNum)%DesCoolVolFlow

```

```

        FanMinAirFlowRate = CalcSysSizing(CurSysNum)%SysAirMinFlowRat *
CalcSysSizing(CurSysNum)%DesCoolVolFlow
        CASE(Heating)
            Fan(FanNum)%MaxAirFlowRate = FinalSysSizing(CurSysNum)%DesHeatVolFlow
            FanMinAirFlowRate = CalcSysSizing(CurSysNum)%SysAirMinFlowRat *
CalcSysSizing(CurSysNum)%DesHeatVolFlow
        CASE(Other)
            Fan(FanNum)%MaxAirFlowRate = FinalSysSizing(CurSysNum)%DesMainVolFlow
            FanMinAirFlowRate = CalcSysSizing(CurSysNum)%SysAirMinFlowRat *
CalcSysSizing(CurSysNum)%DesMainVolFlow
        CASE DEFAULT
            Fan(FanNum)%MaxAirFlowRate = FinalSysSizing(CurSysNum)%DesMainVolFlow
            FanMinAirFlowRate = CalcSysSizing(CurSysNum)%SysAirMinFlowRat *
CalcSysSizing(CurSysNum)%DesMainVolFlow
        END SELECT

    ELSE IF (CurZoneEqNum > 0) THEN

        CALL CheckZoneSizing('FAN:' // TRIM(Fan(FanNum)%FanType) // ':' //
TRIM(Fan(FanNum)%Control), &
            Fan(FanNum)%FanName)
        IF (.NOT. ZoneHeatingOnlyFan) THEN
            Fan(FanNum)%MaxAirFlowRate = MAX(FinalZoneSizing(CurZoneEqNum)%DesCoolVolFlow, &
                FinalZoneSizing(CurZoneEqNum)%DesHeatVolFlow)
        ELSE
            Fan(FanNum)%MaxAirFlowRate = FinalZoneSizing(CurZoneEqNum)%DesHeatVolFlow
        END IF

    END IF

    IF (Fan(FanNum)%MaxAirFlowRate < SmallAirVolFlow) THEN
        Fan(FanNum)%MaxAirFlowRate = 0.0
    END IF

    CALL ReportSizingOutput('FAN:' // TRIM(Fan(FanNum)%FanType) // ':' //
TRIM(Fan(FanNum)%Control), &
        Fan(FanNum)%FanName, 'Max Flow Rate [m3/s]',
Fan(FanNum)%MaxAirFlowRate)

    IF (Fan(FanNum)%Control == 'VARIABLEVOLUME') THEN
        CALL CheckSysSizing('FAN:' // TRIM(Fan(FanNum)%FanType) // ':' // TRIM(Fan(FanNum)%Control),
&
            Fan(FanNum)%FanName)
        Fan(FanNum)%MinAirFlowRate = FanMinAirFlowRate
        CALL ReportSizingOutput('FAN:' // TRIM(Fan(FanNum)%FanType) // ':' //
TRIM(Fan(FanNum)%Control), &
            Fan(FanNum)%FanName, 'Min Flow Rate [m3/s]',
Fan(FanNum)%MinAirFlowRate)
    END IF

    END IF

    RETURN

END SUBROUTINE SizeFan

! End Initialization Section of the Module
!*****

! Begin Algorithm Section of the Module
!*****
SUBROUTINE SimSimpleFan(FanNum)

    ! SUBROUTINE INFORMATION:
    !     AUTHOR      Unknown
    !     DATE WRITTEN Unknown
    !     MODIFIED    na
    !     RE-ENGINEERED na

    ! PURPOSE OF THIS SUBROUTINE:

```

```

! This subroutine simulates the simple constant volume fan.

! METHODOLOGY EMPLOYED:
! Converts design pressure rise and efficiency into fan power and temperature rise
! Constant fan pressure rise is assumed.

! REFERENCES:
! ASHRAE HVAC 2 Toolkit, page 2-3 (FANSIM)

! USE STATEMENTS:
! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

! SUBROUTINE ARGUMENT DEFINITIONS:
Integer, Intent(IN) :: FanNum

! SUBROUTINE PARAMETER DEFINITIONS:
! na

! INTERFACE BLOCK SPECIFICATIONS
! na

! DERIVED TYPE DEFINITIONS
! na

! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
Real RhoAir
Real DeltaPress ! [N/M^2]
Real FanEff
Real MassFlow ! [kg/sec]
Real Tin ! [C]
Real Win
Real FanShaftPower ! power delivered to fan shaft
Real PowerLossToAir ! fan and motor loss to air stream (watts)

DeltaPress = Fan(FanNum)%DeltaPress
FanEff = Fan(FanNum)%FanEff

! For a Constant Volume Simple Fan the Max Flow Rate is the Flow Rate for the fan
Tin = Fan(FanNum)%InletAirTemp
Win = Fan(FanNum)%InletAirHumRat
RhoAir = Fan(FanNum)%RhoAirStdInit
MassFlow = MIN(Fan(FanNum)%InletAirMassFlowRate, Fan(FanNum)%MaxAirMassFlowRate)
MassFlow = MAX(MassFlow, Fan(FanNum)%MinAirMassFlowRate)
!
! Determine the Fan Schedule for the Time step
If( ( GetCurrentScheduleValue(Fan(FanNum)%SchedPtr)>0.0 .and. Massflow>0.0 .or. TurnFansOn .and.
Massflow>0.0) &
.and. .NOT.TurnFansOff ) Then
! Fan is operating
Fan(FanNum)%FanPower = MassFlow*DeltaPress/(FanEff*RhoAir) ! total fan power
FanShaftPower = Fan(FanNum)%MotEff * Fan(FanNum)%FanPower ! power delivered to shaft
PowerLossToAir = FanShaftPower + (Fan(FanNum)%FanPower - FanShaftPower) *
Fan(FanNum)%MotInAirFrac
Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy + PowerLossToAir/MassFlow
! This fan does not change the moisture or Mass Flow across the component
Fan(FanNum)%OutletAirHumRat = Fan(FanNum)%InletAirHumRat
Fan(FanNum)%OutletAirMassFlowRate = MassFlow
Fan(FanNum)%OutletAirTemp =
PsyTdbFnHW(Fan(FanNum)%OutletAirEnthalpy, Fan(FanNum)%OutletAirHumRat)
Else
! Fan is off and not operating no power consumed and mass flow rate.
Fan(FanNum)%FanPower = 0.0
FanShaftPower = 0.0
PowerLossToAir = 0.0
Fan(FanNum)%OutletAirMassFlowRate = 0.0
Fan(FanNum)%OutletAirHumRat = Fan(FanNum)%InletAirHumRat
Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy
Fan(FanNum)%OutletAirTemp = Fan(FanNum)%InletAirTemp

```

```

! Set the Control Flow variables to 0.0 flow when OFF.
Fan(FanNum)%MassFlowRateMaxAvail = 0.0
Fan(FanNum)%MassFlowRateMinAvail = 0.0

End If

RETURN
END SUBROUTINE SimSimpleFan

SUBROUTINE SimVariableVolumeFan(FanNum)

! SUBROUTINE INFORMATION:
!   AUTHOR           Unknown
!   DATE WRITTEN      Unknown
!   MODIFIED          Phil Haves
!   RE-ENGINEERED     na

! PURPOSE OF THIS SUBROUTINE:
! This subroutine simulates the simple variable volume fan.

! METHODOLOGY EMPLOYED:
! Converts design pressure rise and efficiency into fan power and temperature rise
! Constant fan pressure rise is assumed.
! Uses curves of fan power fraction vs. fan part load to determine fan power at
! off design conditions.

! REFERENCES:
! ASHRAE HVAC 2 Toolkit, page 2-3 (FANSIM)

! USE STATEMENTS:
! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

! SUBROUTINE ARGUMENT DEFINITIONS:
Integer, Intent(IN) :: FanNum

! SUBROUTINE PARAMETER DEFINITIONS:
! na

! INTERFACE BLOCK SPECIFICATIONS
! na

! DERIVED TYPE DEFINITIONS
! na

! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
Real RhoAir
Real DeltaPress ! [N/M^2 = Pa]
Real FanEff     ! Total fan efficiency - combined efficiency of fan, drive train,
                ! motor and variable speed controller (if any)
Real MassFlow   ! [kg/sec]
Real Tin        ! [C]
Real Win
Real PartLoadFrac
REAL MaxFlowFrac !Variable Volume Fan Max Flow Fraction [-]
REAL MinFlowFrac !Variable Volume Fan Min Flow Fraction [-]
REAL FlowFrac    !Variable Volume Fan Flow Fraction [-]
Real FanShaftPower ! power delivered to fan shaft
Real PowerLossToAir ! fan and motor loss to air stream (watts)

! Simple Variable Volume Fan - default values from DOE-2
! Type of Fan      Coeff1      Coeff2      Coeff3      Coeff4      Coeff5
! INLET VANE DAMPERS 0.35071223 0.30850535 -0.54137364 0.87198823 0.000
! DISCHARGE DAMPERS 0.37073425 0.97250253 -0.34240761 0.000      0.000
! VARIABLE SPEED MOTOR 0.0015302446 0.0052080574 1.1086242 -0.11635563 0.000

DeltaPress = Fan(FanNum)%DeltaPress
FanEff      = Fan(FanNum)%FanEff

```



```

Tin      = Fan(FanNum)%InletAirTemp
Win      = Fan(FanNum)%InletAirHumRat
RhoAir   = Fan(FanNum)%RhoAirStdInit
MassFlow = MIN(Fan(FanNum)%InletAirMassFlowRate, Fan(FanNum)%MaxAirMassFlowRate)
! MassFlow = MAX(MassFlow, Fan(FanNum)%MinAirMassFlowRate)

! Calculate and check limits on fraction of system flow
MaxFlowFrac = 1.0
! MinFlowFrac is calculated from the ration of the volume flows and is non-dimensional
MinFlowFrac = Fan(FanNum)%MinAirFlowRate/Fan(FanNum)%MaxAirFlowRate
! The actual flow fraction is calculated from MassFlow and the MaxVolumeFlow * AirDensity
FlowFrac = MassFlow/(Fan(FanNum)%MaxAirMassFlowRate)

! Calculate the part Load Fraction (PH 7/13/03)

FlowFrac = MAX(MinFlowFrac, MIN(FlowFrac, 1.0)) ! limit flow fraction to allowed range

PartLoadFrac = Fan(FanNum)%FanCoeff(1) + Fan(FanNum)%FanCoeff(2)*FlowFrac + &
Fan(FanNum)%FanCoeff(3)*FlowFrac**2 + Fan(FanNum)%FanCoeff(4)*FlowFrac**3 + &
Fan(FanNum)%FanCoeff(5)*FlowFrac**4

! Determine the Fan Schedule for the Time step
If( ( GetCurrentScheduleValue(Fan(FanNum)%SchedPtr)>0.0 .and. Massflow>0.0 .or. TurnFansOn .and.
Massflow>0.0) &
.and. .NOT.TurnFansOff ) Then
! Fan is operating - calculate power loss and enthalpy rise
! Fan(FanNum)%FanPower = PartLoadFrac*FullMassFlow*DeltaPress/(FanEff*RhoAir) ! total fan power
Fan(FanNum)%FanPower = PartLoadFrac*Fan(FanNum)%MaxAirMassFlowRate*DeltaPress/(FanEff*RhoAir) !
total fan power (PH 7/13/03)
FanShaftPower = Fan(FanNum)%MotEff * Fan(FanNum)%FanPower ! power delivered to shaft
PowerLossToAir = FanShaftPower + (Fan(FanNum)%FanPower - FanShaftPower) *
Fan(FanNum)%MotInAirFrac
Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy + PowerLossToAir/MassFlow
! This fan does not change the moisture or Mass Flow across the component
Fan(FanNum)%OutletAirHumRat = Fan(FanNum)%InletAirHumRat
Fan(FanNum)%OutletAirMassFlowRate = MassFlow
Fan(FanNum)%OutletAirTemp =
PsyTdbFnHW(Fan(FanNum)%OutletAirEnthalpy, Fan(FanNum)%OutletAirHumRat)
Else
! Fan is off and not operating no power consumed and mass flow rate.
Fan(FanNum)%FanPower = 0.0
FanShaftPower = 0.0
PowerLossToAir = 0.0
Fan(FanNum)%OutletAirMassFlowRate = 0.0
Fan(FanNum)%OutletAirHumRat = Fan(FanNum)%InletAirHumRat
Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy
Fan(FanNum)%OutletAirTemp = Fan(FanNum)%InletAirTemp
! Set the Control Flow variables to 0.0 flow when OFF.
Fan(FanNum)%MassFlowRateMaxAvail = 0.0
Fan(FanNum)%MassFlowRateMinAvail = 0.0
End If

RETURN
END SUBROUTINE SimVariableVolumeFan

SUBROUTINE SimOnOffFan(FanNum)

! SUBROUTINE INFORMATION:
!   AUTHOR      Unknown
!   DATE WRITTEN Unknown
!   MODIFIED    Shirey, May 2001
!   RE-ENGINEERED na

! PURPOSE OF THIS SUBROUTINE:
! This subroutine simulates the simple on/off fan.

! METHODOLOGY EMPLOYED:
! Converts design pressure rise and efficiency into fan power and temperature rise
! Constant fan pressure rise is assumed.
! Uses curves of fan power fraction vs. fan part load to determine fan power at

```

```

! off design conditions.
! Same as simple (constant volume) fan, except added part-load curve input

! REFERENCES:
! ASHRAE HVAC 2 Toolkit, page 2-3 (FANSIM)

! USE STATEMENTS:
USE CurveManager, ONLY: CurveValue

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

! SUBROUTINE ARGUMENT DEFINITIONS:
Integer, Intent(IN) :: FanNum

! SUBROUTINE PARAMETER DEFINITIONS:
! na

! INTERFACE BLOCK SPECIFICATIONS
! na

! DERIVED TYPE DEFINITIONS
! na

! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
Real RhoAir
Real DeltaPress ! [N/M^2]
Real FanEff
Real MassFlow   ! [kg/sec]
Real Tin        ! [C]
Real Win
Real PartLoadRatio !Ratio of actual mass flow rate to max mass flow rate
REAL FlowFrac     !Actual Fan Flow Fraction = actual mass flow rate / max air mass flow
rate
Real FanShaftPower ! power delivered to fan shaft
Real PowerLossToAir ! fan and motor loss to air stream (watts)

DeltaPress = Fan(FanNum)%DeltaPress
FanEff      = Fan(FanNum)%FanEff

Tin         = Fan(FanNum)%InletAirTemp
Win         = Fan(FanNum)%InletAirHumRat
RhoAir      = Fan(FanNum)%RhoAirStdInit
MassFlow    = MIN(Fan(FanNum)%InletAirMassFlowRate,Fan(FanNum)%MaxAirMassFlowRate)
MassFlow    = MAX(MassFlow,Fan(FanNum)%MinAirMassFlowRate)
Fan(FanNum)%FanRuntimeFraction = 0.0

! The actual flow fraction is calculated from MassFlow and the MaxVolumeFlow * AirDensity
FlowFrac = MassFlow/(Fan(FanNum)%MaxAirMassFlowRate)

! Calculate the part load ratio, can't be greater than 1
PartLoadRatio= MIN(1.0,FlowFrac)
! Determine the Fan Schedule for the Time step
IF( ( GetCurrentScheduleValue(Fan(FanNum)%SchedPtr)>0.0 .and. Massflow>0.0 .or. TurnFansOn .and.
Massflow>0.0) &
.and. .NOT.TurnFansOff ) THEN
! Fan is operating
IF (OnOffFanPartLoadFraction <= 0.0) THEN
CALL ShowWarningError('FAN:SIMPLE:ONOFF, OnOffFanPartLoadFraction <= 0.0, Reset to 1.0')
OnOffFanPartLoadFraction = 1.0 ! avoid divide by zero or negative PLF
END IF

IF (OnOffFanPartLoadFraction < 0.7) THEN
OnOffFanPartLoadFraction = 0.7 ! a warning message is already issued from the DX coils or
gas heating coil
END IF
! Keep fan runtime fraction between 0.0 and 1.0
Fan(FanNum)%FanRuntimeFraction = MAX(0.0,MIN(1.0,PartLoadRatio/OnOffFanPartLoadFraction))
! Fan(FanNum)%FanPower = MassFlow*DeltaPress/(FanEff*RhoAir*OnOffFanPartLoadFraction)! total
fan power

```

```

    Fan(FanNum)%FanPower =
Fan(FanNum)%MaxAirMassFlowRate*Fan(FanNum)%FanRuntimeFraction*DeltaPress/(FanEff*RhoAir)!total fan
power
    ! OnOffFanPartLoadFraction is passed via DataHVACGlobals from the cooling or heating coil that
is
    ! requesting the fan to operate in cycling fan/cycling coil mode
    OnOffFanPartLoadFraction = 1.0 ! reset to 1 in case other on/off fan is called without a part
load curve
    FanShaftPower = Fan(FanNum)%MotEff * Fan(FanNum)%FanPower ! power delivered to shaft
    PowerLossToAir = FanShaftPower + (Fan(FanNum)%FanPower - FanShaftPower) *
Fan(FanNum)%MotInAirFrac
    Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy + PowerLossToAir/MassFlow
    ! This fan does not change the moisture or Mass Flow across the component
    Fan(FanNum)%OutletAirHumRat = Fan(FanNum)%InletAirHumRat
    Fan(FanNum)%OutletAirMassFlowRate = MassFlow
    ! Fan(FanNum)%OutletAirTemp = Tin + PowerLossToAir/(MassFlow*PsyCpAirFnWTdb(Win,Tin))
    Fan(FanNum)%OutletAirTemp =
PsyTdbFnHW(Fan(FanNum)%OutletAirEnthalpy,Fan(FanNum)%OutletAirHumRat)
ELSE
    ! Fan is off and not operating no power consumed and mass flow rate.
    Fan(FanNum)%FanPower = 0.0
    FanShaftPower = 0.0
    PowerLossToAir = 0.0
    Fan(FanNum)%OutletAirMassFlowRate = 0.0
    Fan(FanNum)%OutletAirHumRat = Fan(FanNum)%InletAirHumRat
    Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy
    Fan(FanNum)%OutletAirTemp = Fan(FanNum)%InletAirTemp
    ! Set the Control Flow variables to 0.0 flow when OFF.
    Fan(FanNum)%MassFlowRateMaxAvail = 0.0
    Fan(FanNum)%MassFlowRateMinAvail = 0.0
END IF

RETURN
END SUBROUTINE SimOnOffFan

SUBROUTINE SimZoneExhaustFan(FanNum)

    ! SUBROUTINE INFORMATION:
    !     AUTHOR          Fred Buhl
    !     DATE WRITTEN    Jan 2000
    !     MODIFIED        na
    !     RE-ENGINEERED   na

    ! PURPOSE OF THIS SUBROUTINE:
    ! This subroutine simulates the Zone Exhaust Fan

    ! METHODOLOGY EMPLOYED:
    ! Converts design pressure rise and efficiency into fan power and temperature rise
    ! Constant fan pressure rise is assumed.

    ! REFERENCES:
    ! ASHRAE HVAC 2 Toolkit, page 2-3 (FANSIM)

    ! USE STATEMENTS:
    ! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

    ! SUBROUTINE ARGUMENT DEFINITIONS:
    Integer, Intent(IN) :: FanNum

    ! SUBROUTINE PARAMETER DEFINITIONS:
    ! na

    ! INTERFACE BLOCK SPECIFICATIONS
    ! na

    ! DERIVED TYPE DEFINITIONS
    ! na

```

```

      ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
      Real RhoAir
      Real DeltaPress ! [N/M^2]
      Real FanEff
      Real MassFlow ! [kg/sec]
      Real Tin ! [C]
      Real Win
      Real PowerLossToAir ! fan and motor loss to air stream (watts)

      DeltaPress = Fan(FanNum)%DeltaPress
      FanEff = Fan(FanNum)%FanEff

      ! For a Constant Volume Simple Fan the Max Flow Rate is the Flow Rate for the fan
      Tin = Fan(FanNum)%InletAirTemp
      Win = Fan(FanNum)%InletAirHumRat
      RhoAir = Fan(FanNum)%RhoAirStdInit
      MassFlow = Fan(FanNum)%InletAirMassFlowRate
      !
      ! Determine the Fan Schedule for the Time step
      If( ( GetCurrentScheduleValue(Fan(FanNum)%SchedPtr)>0.0 .or. TurnFansOn ) &
         .and. .NOT.TurnFansOff ) Then
         ! Fan is operating
         Fan(FanNum)%FanPower = MassFlow*DeltaPress/(FanEff*RhoAir) ! total fan power
         PowerLossToAir = Fan(FanNum)%FanPower
         Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy + PowerLossToAir/MassFlow
         ! This fan does not change the moisture or Mass Flow across the component
         Fan(FanNum)%OutletAirHumRat = Fan(FanNum)%InletAirHumRat
         Fan(FanNum)%OutletAirMassFlowRate = MassFlow
         Fan(FanNum)%OutletAirTemp =
         PsyTdbFnHW(Fan(FanNum)%OutletAirEnthalpy,Fan(FanNum)%OutletAirHumRat)
      Else
         ! Fan is off and not operating no power consumed and mass flow rate.
         Fan(FanNum)%FanPower = 0.0
         PowerLossToAir = 0.0
         Fan(FanNum)%OutletAirMassFlowRate = 0.0
         Fan(FanNum)%OutletAirHumRat = Fan(FanNum)%InletAirHumRat
         Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy
         Fan(FanNum)%OutletAirTemp = Fan(FanNum)%InletAirTemp
         ! Set the Control Flow variables to 0.0 flow when OFF.
         Fan(FanNum)%MassFlowRateMaxAvail = 0.0
         Fan(FanNum)%MassFlowRateMinAvail = 0.0
         Fan(FanNum)%InletAirMassFlowRate = 0.0

      End If

      RETURN
END SUBROUTINE SimZoneExhaustFan

! End Algorithm Section of the Module
! *****

! Beginning of Update subroutines for the Fan Module
! *****

SUBROUTINE UpdateFan(FanNum)

      ! SUBROUTINE INFORMATION:
      !       AUTHOR      Richard Liesen
      !       DATE WRITTEN April 1998
      !       MODIFIED    na
      !       RE-ENGINEERED na

      ! PURPOSE OF THIS SUBROUTINE:
      ! This subroutine updates the fan outlet nodes.

      ! METHODOLOGY EMPLOYED:
      ! Data is moved from the fan data structure to the fan outlet nodes.

      ! REFERENCES:
      ! na

```

```

        ! USE STATEMENTS:
        ! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

        ! SUBROUTINE ARGUMENT DEFINITIONS:
Integer, Intent(IN) :: FanNum

        ! SUBROUTINE PARAMETER DEFINITIONS:
        ! na

        ! INTERFACE BLOCK SPECIFICATIONS
        ! na

        ! DERIVED TYPE DEFINITIONS
        ! na

        ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
Integer          :: OutletNode
Integer          :: InletNode

OutletNode = Fan(FanNum)%OutletNodeNum
InletNode = Fan(FanNum)%InletNodeNum

! Set the outlet air nodes of the fan
Node(OutletNode)%MassFlowRate = Fan(FanNum)%OutletAirMassFlowRate
Node(OutletNode)%Temp         = Fan(FanNum)%OutletAirTemp
Node(OutletNode)%HumRat       = Fan(FanNum)%OutletAirHumRat
Node(OutletNode)%Enthalpy     = Fan(FanNum)%OutletAirEnthalpy
! Set the outlet nodes for properties that just pass through & not used
Node(OutletNode)%Quality      = Node(InletNode)%Quality
Node(OutletNode)%Press        = Node(InletNode)%Press

! Set the Node Flow Control Variables from the Fan Control Variables
Node(OutletNode)%MassFlowRateMaxAvail = Fan(FanNum)%MassFlowRateMaxAvail
Node(OutletNode)%MassFlowRateMinAvail = Fan(FanNum)%MassFlowRateMinAvail

IF (Fan(FanNum)%FanType .EQ. 'ZONE EXHAUST FAN') THEN
    Node(InletNode)%MassFlowRate = Fan(FanNum)%InletAirMassFlowRate
END IF

RETURN
END Subroutine UpdateFan

!      End of Update subroutines for the Fan Module
! *****

! Beginning of Reporting subroutines for the Fan Module
! *****

SUBROUTINE ReportFan(FanNum)

        ! SUBROUTINE INFORMATION:
        !      AUTHOR      Richard Liesen
        !      DATE WRITTEN April 1998
        !      MODIFIED    na
        !      RE-ENGINEERED na

        ! PURPOSE OF THIS SUBROUTINE:
        ! This subroutine updates the report variables for the fans.

        ! METHODOLOGY EMPLOYED:
        ! na

        ! REFERENCES:
        ! na

        ! USE STATEMENTS:

```

```

Use DataHVACGlobals, ONLY: TimeStepSys, FanElecPower

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

      ! SUBROUTINE ARGUMENT DEFINITIONS:
      Integer, Intent(IN) :: FanNum

      ! SUBROUTINE PARAMETER DEFINITIONS:
      ! na

      ! INTERFACE BLOCK SPECIFICATIONS
      ! na

      ! DERIVED TYPE DEFINITIONS
      ! na

      ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
      ! na

      Fan(FanNum)%FanEnergy=Fan(FanNum)%FanPower*TimeStepSys*3600
      Fan(FanNum)%DeltaTemp=Fan(FanNum)%OutletAirTemp - Fan(FanNum)%InletAirTemp
      FanElecPower = Fan(FanNum)%FanPower

RETURN
END Subroutine ReportFan

!      End of Reporting subroutines for the Fan Module
! *****

!      NOTICE
!
!      Copyright © 1996-2004 The Board of Trustees of the University of Illinois
!      and The Regents of the University of California through Ernest Orlando Lawrence
!      Berkeley National Laboratory. All rights reserved.
!
!      Portions of the EnergyPlus software package have been developed and copyrighted
!      by other individuals, companies and institutions. These portions have been
!      incorporated into the EnergyPlus software package under license. For a complete
!      list of contributors, see "Notice" located in EnergyPlus.f90.
!
!      NOTICE: The U.S. Government is granted for itself and others acting on its
!      behalf a paid-up, nonexclusive, irrevocable, worldwide license in this data to
!      reproduce, prepare derivative works, and perform publicly and display publicly.
!      Beginning five (5) years after permission to assert copyright is granted,
!      subject to two possible five year renewals, the U.S. Government is granted for
!      itself and others acting on its behalf a paid-up, non-exclusive, irrevocable
!      worldwide license in this data to reproduce, prepare derivative works,
!      distribute copies to the public, perform publicly and display publicly, and to
!      permit others to do so.
!
!      TRADEMARKS: EnergyPlus is a trademark of the US Department of Energy.
!
End Module Fans

```

This example can be used as a template for new HVAC component modules. In particular, the commenting structure in the module and within the subroutines should be followed closely. Of course, there is no perfect example module – this one is particularly simple. Some others that might be examined are in files Humidifiers.f90, HVACHeatingCoils.f90 and PlantChillers.f90. Templates are also available as separate files.

How it fits together

Although we have designed the EnergyPlus modules to be as independent as possible, obviously they cannot be completely independent. How does an EnergyPlus HVAC module fit in with the rest of the program? First, we will show some subroutine calling trees that will display the overall program structure.

Top Level Calling Tree

EnergyPlus

- ProcessInput (in InputProcessor)
- ManageSimulation (in SimulationManager)
 - ManageWeather (in WeatherManager)
 - ManageHeatBalance (in HeatBalanceManager)
 - ManageSurfaceHeatBalance (in HeatBalanceSurfaceManager)
 - ManageAirHeatBalance (in HeatBalanceAirManager)
 - CalcHeatBalanceAir (in HeatBalanceAirManager)
 - ManageHVAC (in HVACManager)

The HVAC part of EnergyPlus is divided into a number of simulation blocks. At this point, there are blocks for the air system, the zone equipment, the plant supply, the plant demand, the condenser supply, and the condenser demand. There will be simulation blocks for waste heat supply and usage as well as electricity and gas. Within each HVAC time step, the blocks are simulated repeatedly until the conditions on each side of each block interface match up. The following calling tree represents the high level HVAC simulation structure. It is schematic – not all routines are shown.

High Level HVAC Calling Tree (schematic – not all routines are shown)

ManageHVAC (in HVACManager)

- ZoneAirUpdate('PREDICT', . . .) (in HVACManager)
estimate the zone heating or cooling demand
- SimHVAC (in HVACManager)
 - ManageSetPoints (in SetPointManager)
 - SimSelectedEquipment (in HVACManager)
 - ManageAirLoops (in SimAirServingZones)
 - ManageZoneEquipment (in ZoneEquipmentManager)
 - ManagePlantSupplySides (in PlantLoopSupplySideManager)
 - ManagePlantDemandSides (in PlantdemandSideLoops)
 - ManageCondSupplySides (in CondLoopManager)
 - ManageCondenserDemandSides (in CondenserDemandSideLoops)
- ZoneAirUpdate('CORRECT', . . .) (in HVACManager)
From the amount of heating and cooling actually provided by the HVAC system, calculate the zone temperatures.

Each of the “Manage” routines has a different structure, since the simulation to be performed is different in each case. We will show schematic calling trees for several of the “Manage” routines.

Air System Calling Tree (schematic – not all routines are shown)

ManageAirLoops (in SimAirServingZones)

- GetAirPathData (in SimAirServingZones)
- InitAirLoops (in SimAirServingZones)
- SimAirLoops (in SimAirServingZones)
 - SimAirLoopComponent (in SimAirServingZones)
 - UpdateBranchConnections (in SimAirServingZones)
 - ManageOutsideAirSystem (in MixedAir)
 - SimOutsideAirSys (in MixedAir)
 - SimOAController (in MixedAir)
 - SimOACComponent (in Mixed Air)
 - SimOAMixer (in MixedAir)
 - SimulateFanComponents(in FanSimulation; file HVACFanComponent)
 - SimulateWaterCoilComponents (in WaterCoilSimulation; file HVACWaterCoilComponent)
 - SimHeatRecovery (in HeatRecovery)
 - SimDesiccantDehumidifier (in DesiccantDehumidifiers)
 - SimulateFanComponents (in FanSimulation; file HVACFanComponent)
 - SimulateWaterCoilComponents (in WaterCoilSimulation; file HVACWaterCoilComponent)
 - SimulateHeatingCoilComponents (in HeatingCoils; file HVACHeatingCoils)
 - SimDXCoolingSystem (in HVACDXSystem)
 - SimFurnace (in Furnaces; file HVACFurnace)
 - SimHumidifier (in Humidifiers)
 - SimEvapCooler (in EvaporativeCoolers; file HVACEvapComponent)
 - SimDesiccantDehumidifier (in DesiccantDehumidifiers)
 - SimHeatRecovery (in HeatRecovery)
 - ManageControllers (in Controllers)
 - GetControllerInput (in Controllers)
 - InitController (in Controllers)
 - SimpleController (in Controllers)
 - LimitController (in Controllers)
 - UpdateController (in Controllers)
 - Report Controller (in Controllers)
 - ResolveSysFlow (in SimAirServingZones)
 - UpdateHVACInterface (in HVACInterfaceManager)
 - ReportAirLoops (in SimAirServingZones)

Plant Supply Calling Tree (schematic – not all routines are shown)

ManagePlantSupplySides (in PlantLoopSupplySideManager)

- GetLoopData (in PlantLoopSupplySideManager)
- SetLoopInitialConditions (in PlantLoopSupplySideManager)

- CalcLoopDemand (in PlantLoopSupplySideManager)
- ManagePlantLoopOperation (in PlantCondLoopOperation)
- DistributeLoad (in PlantLoopSupplySideManager)
- SimPlantEquip (in PlantLoopSupplySideManager)
 - SimPipes (in Pipes; file PlantPipes)
 - SimPumps (in Pumps; file PlantPumps)
 - SimEngineDrivenChiller (in ChillerEngineDriven ; file PlantChillers)
 - SimBLASTAbsorber (in ChillerAbsorption ; file PlantAbsorptionChillers)
 - SimElectricChiller (in ChillerElectric ; file PlantChillers)
 - SimGTChiller (in ChillerGasTurbine ; file PlantChillers)
 - SimConstCOPChiller (in ChillerConstCOP; file PlantChillers)
 - SimBLASTChiller (in ChillerBLAST ; file PlantChillers)
 - SimOutsideCooling (in OutsideCoolingSources ; file PlantOutsideCoolingSources)
 - SimGasAbsorber (in ChillerGasAbsorption ; file PlantGasAbsorptionChiller)
 - SimBoiler (in Boilers; file PlantBoilers)
 - SimWaterHeater (in WaterHeaters ; file PlantWaterHeater)
 - SimOutsideHeating (in OutsideHeatingSources; file PlantOutsideHeatingSources)
- UpdateSplitter (in PlantLoopSupplySideManager)
- SolveFlowNetwork (in PlantLoopSupplySideManager)
- CalcLoopDemand (in PlantLoopSupplySideManager)
- SimPlantEquip (in PlantLoopSupplySideManager)
- UpdateSplitter
- UpdateMixer (in PlantLoopSupplySideManager)
- SimPlantEquip (in PlantLoopSupplySideManager)
- CheckLoopExitNodes (in PlantLoopSupplySideManager)
- UpdateHVACInterface (in HVACInterfaceManager)
- UpdateReportVars (in PlantLoopSupplySideManager)

Zone Equipment Calling Tree (schematic – not all routines are shown)

ManageZoneEquipment (in ZoneEquipmentManager)

- GetZoneEquipment (in ZoneEquipmentManager)
- InitZoneEquipment (in ZoneEquipmentManager)
- SimZoneEquioment (in ZoneEquipmentManager)
 - SimAirLoopSplitter (in Splitters; file HVACSplitterComponent)
 - SimAirZonePlenum (in ZonePlenum; file ZonePlenumComponent)
 - SetZoneEquipSimOrder (in ZoneEquipmentManager)
 - InitSystemOutputRequired (in ZoneEquipmentManager)
 - ManageZoneAirLoopEquipment (in ZoneAirLoopEquipmentManager)
 - GetZoneAirLoopEquipment (in ZoneAirLoopEquipmentManager)
 - SimZoneAirLoopEquipment (in ZoneAirLoopEquipmentManager)
 - SimulateDualDuct (in DualDuct; file HVACDualDuctSystem)
 - GetDualDuctInput (in DualDuct; file HVACDualDuctSystem)
 - InitDualDuct (in DualDuct; file HVACDualDuctSystem)
 - SimDualDuctConstVol (in DualDuct; file HVACDualDuctSystem)
 - SimDualDuctVarVol (in DualDuct; file HVACDualDuctSystem)
 - UpdateDualDuct (in DualDuct; file HVACDualDuctSystem)

- ReportDualDuct (in DualDuct; file HVACDualDuctSystem)
- SimulateSingleDuct (in SingleDuct; file HVACSingleDuctSystem)
 - GetSysInput (in SingleDuct; file HVACSingleDuctSystem)
 - InitSys (in SingleDuct; file HVACSingleDuctSystem)
 - SimConstVol (in SingleDuct; file HVACSingleDuctSystem)
 - SimVAV (in SingleDuct; file HVACSingleDuctSystem)
 - ReportSys (in SingleDuct; file HVACSingleDuctSystem)
- SimPIU (in PoweredInductionUnits)
 - GetPIUs (in PoweredInductionUnits)
 - InitPIUs (in PoweredInductionUnits)
 - CalcSeriesPIU (in PoweredInductionUnits)
 - CalcParallelPIU (in PoweredInductionUnits)
 - ReportPIU (in PoweredInductionUnits)
- SimDirectAir (in DirectAirManager; file DirectAir)
- SimPurchasedAir (in PurchasedAirManager)
- SimWindowAC (in WindowAC)
- SimFanCoilUnit (in FanCoilUnits)
- SimUnitVentilator (in UnitVentilator)
- SimUnitHeater (in UnitHeater)
- SimBaseboard (in BaseboardRadiator)
- SimHighTempRadiantSystem (in HighTempRadiantSystem; file RadiantSystemHighTemp)
- SimLowTempRadiantSystem (in LowTempRadiantSystem; file RadiantSystemLowTemp)
- SimulateFanComponents (in Fans; file HVACFanComponent)
- SimHeatRecovery (in HeatRecovery)
- UpdateSystemOutputRequired (in ZoneEquipmentManager)
- SimAirLoopSplitter (in Splitters; file HVACSplitterComponent)
- SimAirZonePlenum (in ZonePlenum; file ZonePlenumComponent)
- CalcZoneMassBalance (in ZoneEquipmentManager)
- CalcZoneLeavingConditions (in ZoneEquipmentManager)
- SimReturnAirPath (in ReturnAirPathManager; file ReturnAirPath)
 - SimAirMixer (in Mixers; HVACMixerComponent)
 - SimAirZonePlenum (in ZonePlenum; file ZonePlenumComponent)
- RecordZoneEquipment (in ZoneEquipmentManager)
- ReportZoneEquipment (in ZoneEquipmentManager)

Inserting the New Module into the Program

Let us return to our example new module NewHVACComponent. Since the module does its own input and output, adding the NewHVACComponent model to the program simply means adding a call to the driver routine SimNewHVACComponent from the correct place in EnergyPlus. In the simplest case, there is only one location from which the driver routine should be called. In some cases, though, more than one HVAC simulation block will need to use the new component model.

SimulateWaterCoilComponents, for instance, can be used in both zone equipment and air systems for heating, reheating and cooling coils. In the air system simulation it is called from two places: the main air system simulation, and the mixed air simulation – the outside air duct might contain a separate cooling coil.

Let us assume that the NewHVACComponent will be part of the air system – perhaps it is a solid desiccant wheel. Examining the air system calling tree we see that

SimAirLoopComponent is one routine that will invoke the new component, and - if we want the component to possibly be in the outside air stream – then SimOACComponent is the other routine that will need to call the new component simulation. Generally, all that is involved is adding a new CASE statement to a Fortran SELECT construct. For instance in SimAirLoopComponent this would look like:

```

SELECT CASE(CompType)

CASE('OUTSIDE AIR SYSTEM')
    CALL ManageOutsideAirSystem(CompName,FirstHVACIteration,LastSim)

! Fan Types for the air sys simulation
CASE('FAN:SIMPLE:CONSTVOLUME')
    CALL SimulateFanComponents(CompName,FirstHVACIteration)
CASE('FAN:SIMPLE:VARIABLEVOLUME')
    CALL SimulateFanComponents(CompName,FirstHVACIteration)

! Coil Types for the air sys simulation
CASE('COIL:DX:COOLINGHEATEXCHANGERASSISTED')
    CALL SimHXAssistedCoolingCoil(CompName,FirstHVACIteration,On,0.0)
CASE('COIL:WATER:COOLINGHEATEXCHANGERASSISTED')
    CALL SimHXAssistedCoolingCoil(CompName,FirstHVACIteration,On,0.0)
CASE('COIL:WATER:SIMPLECOOLING')
    CALL SimulateWaterCoilComponents(CompName,FirstHVACIteration)
CASE('COIL:WATER:SIMPLEHEATING')
    CALL SimulateWaterCoilComponents(CompName,FirstHVACIteration)
CASE('COIL:WATER:DETAILEDFLATCOOLING')
    CALL SimulateWaterCoilComponents(CompName,FirstHVACIteration)
CASE('COIL:ELECTRIC:HEATING')
    CALL SimulateHeatingCoilComponents(CompName=CompName,&
        FirstHVACIteration=FirstHVACIteration,&
        QCoilReq=0.0)
CASE('COIL:GAS:HEATING')
    CALL SimulateHeatingCoilComponents(CompName=CompName,&
        FirstHVACIteration=FirstHVACIteration,&
        QCoilReq=0.0)
CASE('DXSYSTEM:AIRLOOP')
    CALL SimDXCoolingSystem(CompName)

CASE('FURNACE:BLOWTHRU:HEATONLY')
    CALL SimFurnace(CompName, FirstHVACIteration)
CASE('FURNACE:BLOWTHRU:HEATCOOL')
    CALL SimFurnace(CompName, FirstHVACIteration)
CASE('UNITARYSYSTEM:BLOWTHRU:HEATONLY')
    CALL SimFurnace(CompName, FirstHVACIteration)
CASE('UNITARYSYSTEM:BLOWTHRU:HEATCOOL')
    CALL SimFurnace(CompName, FirstHVACIteration)
CASE('UNITARYSYSTEM:HEATPUMP:AIRTOAIR')
    CALL SimFurnace(CompName, FirstHVACIteration, AirLoopNum)
CASE('UNITARYSYSTEM:HEATPUMP:WATERTOAIR')
    CALL SimFurnace(CompName, FirstHVACIteration, AirLoopNum)

! Humidifier Types for the air system simulation
CASE('HUMIDIFIER:STEAM:ELECTRICAL')
    CALL SimHumidifier(CompName,FirstHVACIteration)

! Evap Cooler Types for the air system simulation
CASE('EVAPCOOLER:DIRECT:CELDEKPAD')
    CALL SimEvapCooler(CompName)
CASE('EVAPCOOLER:INDIRECT:CELDEKPAD')
    CALL SimEvapCooler(CompName)
CASE('EVAPCOOLER:INDIRECT:WETCOIL')

```

```

CALL SimEvapCooler(CompName)

! Desiccant Dehumidifier Types for the air system simulation
CASE('DESICCANT DEHUMIDIFIER:SOLID')
  CALL SimDesiccantDehumidifier(CompName,FirstHVACIteration)

! Heat recovery
CASE('HEAT EXCHANGER:AIR TO AIR:FLAT PLATE')
  CALL SimHeatRecovery(CompName,FirstHVACIteration)

! New HVAC Component
CASE ('NEW HVAC COMPONENT')
  CALL SimNewHVACComponent(CompName,FirstHVACIteration)

END SELECT

```

The new code is italicized. Do the same thing in SimOAComponent and you are done! Note that “NEW HVAC COMPONENT” is the class name (keyword) for the new component in the IDD file. The class names are converted to upper case in EnergyPlus, so the CASE statement must have the class name in upper case. The actual class name on the IDD file would probably be “New HVAC Component”.

If the new HVAC component is a piece of zone equipment – a cooled beam system, for instance – then the zone equipment calling tree indicates that the call to SimNewHVACComponent would be in SimZoneEquipment. If the new component is a gas fired absorption chiller, the call would be in SimPlantEquip.

In every case, since NewHVACComponent is a new module, a USE statement must be added to the calling subroutine. For instance in SimAirLoopComponent this would look like:

```

SUBROUTINE SimAirLoopComponent(CompType, CompName, FirstHVACIteration,
LastSim)

! SUBROUTINE INFORMATION
!       AUTHOR: Russ Taylor, Dan Fisher, Fred Buhl
!       DATE WRITTEN: Oct 1997
!       MODIFIED: Dec 1997 Fred Buhl
!       RE-ENGINEERED: This is new code, not reengineered

! PURPOSE OF THIS SUBROUTINE:
! Calls the individual air loop component simulation
routines

! METHODOLOGY EMPLOYED: None

! REFERENCES: None

! USE Statements
USE Fans,          Only:SimulateFanComponents
USE WaterCoils,    Only:SimulateWaterCoilComponents
USE MixedAir,      Only:ManageOutsideAirSystem
USE NewHVACComponent,    Only:SimNewHVACComponent

```

Considerations for Legacy Codes

Those module developers who are adding to EnergyPlus's capabilities by adapting existing codes into the module structure should take special considerations.

First and foremost, who owns the legacy code that you are adapting? More on this is covered in Appendix C – Submissions and Check-ins. One must be very careful when developing modules to be implemented in the publicly-available version of EnergyPlus.

Legacy codes will typically come with their own input and output structures. In adapting this to use with EnergyPlus, the module developer will usually want to bypass these routines by either embedding the code into EnergyPlus and using input entirely from the IDD/IDF structure OR writing a simple input file to the legacy code. Either of these approaches can be used.

Code Readability vs. Speed of Execution

Programmers throughout time have had to deal with speed of code execution and it's an ongoing concern. However, compilers are pretty smart these days and, often, can produce speedier code for the hardware platform than the programmer can when he or she uses "speed up" tips. The EnergyPlus development team would rather the code be more "readable" to all than to try to outwit the compilers for every platform. First and foremost, the code is the true document of what EnergyPlus does – other documents will try to explain algorithms and such but must really take a back seat to the code itself.

However, many people may read the code – as developers, we should try to make it as readable at first glance as possible. For a true example from the code and a general indication of preferred style, take the case of the zone temperature update equation. In the [engineering](#) document, the form is recognizable and usual:

$$T_z^t = \frac{\sum_{i=1}^{N_{si}} \dot{Q}_i + \sum_{i=1}^{N_{surfaces}} h_i A_i T_{si} + \sum_{i=1}^{N_{zones}} \dot{m}_i C_p T_{zi} + \dot{m}_{inf} C_p T_{\infty} + \dot{m}_{sys} C_p T_{supply} - \left(\frac{C_z}{\delta t} \right) \left(-3T_z^{t-\delta t} + \frac{3}{2}T_z^{t-2\delta t} - \frac{1}{3}T_z^{t-3\delta t} \right)}{\left(\frac{11}{6} \right) \frac{C_z}{\delta t} + \sum_{i=1}^{N_{surfaces}} h_i A + \sum_{i=1}^{N_{zones}} \dot{m}_i C_p + \dot{m}_{inf} C_p + \dot{m}_{sys} C_p}$$

And, this equation appears in the code (ZoneTempPredictorCorrector Module), as:

```
ZT(ZoneNum)= (CoefSumhat + CoefAirrat*(3.0*ZTM1(ZoneNum) - (3.0/2.0)*ZTM2(ZoneNum) &
+ (1./3.)* ZTM3(ZoneNum))) &
/ ((11.0/6.0)*CoefAirrat+CoefSumha)
```

somewhat abbreviated here due to lack of page width but still recognizable from the original. A better version would actually be:

```
ZT(ZoneNum)= (CoefSumhat - CoefAirrat*(-3.0*ZTM1(ZoneNum) + (3.0/2.0)*ZTM2(ZoneNum) &
- (1./3.)* ZTM3(ZoneNum))) &
/ ((11.0/6.0)*CoefAirrat+CoefSumha)
```

whereas the natural tendency of programming would lead to the less readable:

```
ZT(ZoneNum)= (CoefSumhat + CoefAirrat*(3.0*ZTM1(ZoneNum) - 1.5*ZTM2(ZoneNum) + .333333*
ZTM3(ZoneNum))) &
/ (1.83333*CoefAirrat+CoefSumha)
```

The final version is a correct translation (more or less) from the Engineering/usual representation but much harder to look at in code and realize what is being represented.

Speed of Execution

A critical consideration in speed of execution is one of character string comparisons.

These are typically quite slow and should be reduced as much as possible in the core routines (i.e. those that are executed every zone or hvac time step). An alternative to string comparisons is to define module-level integer parameters, equate a string to a

parameter during the initial subroutine call (e.g. GetInput), and then do integer comparisons through the remainder of the calls to the module.

For example, in the module shown previously (Module Fans). The parameters for fan types are set as Integers:

```
!MODULE PARAMETER DEFINITIONS
INTEGER, PARAMETER :: FanType_SimpleConstVolume = 1
INTEGER, PARAMETER :: FanType_SimpleVAV          = 2
INTEGER, PARAMETER :: FanType_SimpleOnOff        = 3
INTEGER, PARAMETER :: FanType_ZoneExhaust        = 4
```

During the GetInput, string types are shown (this is getting these objects):

```
CALL GetObjectItem('FAN:SIMPLE:CONSTVOLUME', &
                  SimpFanNum, AlphArray, &
                  NumAlphas, NumArray, NumNums, IOSTAT)
. . .
Fan(FanNum)%FanName = AlphArray(1)
Fan(FanNum)%FanType = 'SIMPLE'
. . .
Fan(FanNum)%Control = 'CONSTVOLUME'
Fan(FanNum)%FanType_Num = FanType_SimpleConstVolume
```

Then, during the simulation the integer parameters are used:

```
! Calculate the Correct Fan Model with the current FanNum
IF (Fan(FanNum)%FanType_Num == FanType_SimpleConstVolume) THEN
  Call SimSimpleFan(FanNum)
Else IF (Fan(FanNum)%FanType_Num == FanType_SimpleVAV) THEN
  Call SimVariableVolumeFan(FanNum)
Else If (Fan(FanNum)%FanType_Num == FanType_SimpleOnOff) THEN
  Call SimOnOffFan(FanNum)
Else If (Fan(FanNum)%FanType_Num == FanType_ZoneExhaust) THEN
  Call SimZoneExhaustFan(FanNum)
End If
```

This does not detract from code readability at all but you might be amazed at how much speedier execution is with this versus the string comparisons.

EnergyPlus Services

EnergyPlus provides some standard services that make the developer's task much easier. The developer can concentrate on the new simulation algorithm rather than have to deal with details of input file structure, writing output, obtaining scheduled data, and accessing weather variables.

Global Data

We have tried to limit the amount of global data in EnergyPlus development. Two critical data-only modules, however, have been used:

DataGlobals – contains truly global data (such as number of zones, current hour, simulation status flags, interface statements to error and output routines)

DataEnvironment – contains weather data that is global (such as current outdoor dry-bulb temperature, barometric pressure, etc.)

Other global data may be applicable for sets of modules. For example, **DataSurfaces** contains data that is used in the modules that reference surfaces e.g., shadowing calculations, heat balance calculations).

These are used in routines as the examples have illustrated.

Utility Routines/Functions

EnergyPlus supplies an extensive set of routines to help module developers get input, check values, get schedule values, get and check nodes. These are summarized in the following table and in more detail in the following sections. The table indicates the routine/function name, times of most likely use, and the module (if applicable) that you must USE in the code in order to make the routine available to you. Most of the items mentioned in this table are of particular use in "GetInput" processing. A few later tables will highlight similar functions/routines for simulation purposes.

Table 1. Table of Utility Functions

Routine/Function Name	Use during	Module
GetNumObjectsFound	"GetInput" Processing	InputProcessor
GetObjectItem	"GetInput" Processing	InputProcessor
GetObjectDefMaxArgs	"GetInput" Processing	InputProcessor
GetObjectItemNum	"GetInput" Processing	InputProcessor
FindItemInList	"GetInput" Processing (best) though some are currently used in Simulation	InputProcessor
FindItem	"GetInput" Processing	InputProcessor
SameString	"GetInput" Processing	InputProcessor
VerifyName	"GetInput" Processing	InputProcessor
RangeCheck	"GetInput" Processing	InputProcessor
MakeUPPERCase	"GetInput" Processing	InputProcessor
GetOnlySingleNode	"GetInput" Processing	NodeInputManager

GetNodeNums	"GetInput" Processing	NodeInputManager
InitUniqueNodeCheck, CheckUniqueNodes, EndUniqueNodeCheck	"GetInput" Processing	NodeInputManager
SetupCompSets	"GetInput" Processing	NodeInputManager
TestCompSets	"GetInput" Processing	NodeInputManager
GetNewUnitNumber	(automatically retrieve an available unit number)	EXTERNAL integer function
FindUnitNumber	Find a unit number when you know the name of the file	EXTERNAL integer function
FindNumberinList	"GetInput" Processing/Init processing	EXTERNAL integer function
ValidateComponent	"GetInput" Processing	Subroutine CALL
CheckComponent	"GetInput" Processing – like ValidateComponent but doesn't generate error message if failure	Subroutine CALL
CreateSysTimeIntervalString	Simulation – Error Messages	General
TrimSigDigits	Simulation – Error Messages	General
RoundSigDigits	Simulation – Error Messages	General
GetScheduleIndex	"GetInput" Processing	ScheduleManager
GetCurrentScheduleValue	Simulation	ScheduleManager
CheckScheduleValueMinMax	"GetInput" Processing	ScheduleManager
GetScheduleMinValue	"GetInput" Processing	ScheduleManager
GetScheduleMaxValue	"GetInput" Processing	ScheduleManager
GetCurveIndex	"GetInput" Processing	CurveManager
CurveValue	Simulation	CurveManager

Input Services

The module *InputProcessor* processes the input data files (IDFs). It also reads and parses the IDD file. The *InputProcessor* uses the definition lines in the IDD as directives on how to process each input object in the IDF. The *InputProcessor* also turns all alpha strings into all UPPER CASE. Currently, it does nothing else to those strings – so the number of blanks in a string must match what the calculational modules expect. The *InputProcessor* processes all numeric strings into single precision real numbers. Special characters, such as tabs, should *not* be included in the IDF.

The EnergyPlus module *InputProcessor* provides several routines - generically called the "get" routines – that enable the developer to readily access the data for a new module. These routines are made available by including a "USE InputProcessor" statement in the module or in the routine that will use the "get" routines. The GetFanInput subroutine in the example illustrates some of the uses of the "get" routines.

InputProcessor

The following objects use public routines from the *InputProcessor*. To access these, the code has:

```
Use InputProcessor, ONLY: <routine1>, <routine2>
```


Where the <routine> is one or more of the following:

GetNumObjectsFound

This function returns the number of objects in the input belonging to a particular class. In other terms, it returns the number of instances in the input of a particular component.

```
Example:
USE InputProcessor, ONLY: GetNumObjectsFound
---
NumVAVSys = GetNumObjectsFound('SINGLE DUCT:VAV:REHEAT')
```

Here NumVAVSys will contain the number of single duct VAV terminal units in the input data file (IDF). SINGLE DUCT:VAV:REHEAT is the class name or keyword defining VAV terminal unit input on the IDD file.

GetObjectItem

This subroutine is used to obtain the actual alphanumeric and numeric data for a particular object.

Example:

```
USE InputProcessor, ONLY: GetNumObjectsFound, GetObjectNum
---
INTEGER :: SysNum
INTEGER :: SysIndex
INTEGER :: NumAlphas
INTEGER :: NumNums
INTEGER :: IOSTAT
REAL, DIMENSION(5) :: NumArray
CHARACTER(len=MaxNameLength), DIMENSION(8) :: AlphArray
. . . . .
! Flow
NumVAVSys = GetNumObjectsFound('SINGLE DUCT:VAV:REHEAT')
. . . . .
!Start Loading the System Input
DO SysIndex = 1, NumVAVSys
  CALL GetObjectItem('SINGLE DUCT:VAV:REHEAT',SysIndex,AlphArray,&
    NumAlphas,NumArray,NumNums,IOSTAT)
  SysNum = SysIndex
  . . . . .
  Sys(SysNum)%SysName      = AlphArray(1)
  Sys(SysNum)%SysType      = 'SINGLE DUCT:VAV:REHEAT'
  Sys(SysNum)%ReheatComp   = AlphArray(6)
  Sys(SysNum)%ReheatName   = AlphArray(7)
  . . . . .
END DO      ! end Number of Sys Loop      END IF
```

Here GetObjectItem is called with inputs 'SINGLE DUCT:VAV:REHEAT' – the class of object we want to input – and SysIndex – the index of the object on the input file. If SysIndex is 3, the call to GetObjectItem will get the data for the third VAV terminal unit on the input file. Output is returned in the remaining arguments. AlphArray contains in order all the alphanumeric data items for a single VAV terminal unit. NumArray contains all the numeric data items. NumAlphas is the number of alphanumeric items read; NumNums is the number of numeric data items read. IOSTAT is a status flag: -1 means there was an error; +1 means the input was OK. AlphArray and NumArray should be dimensioned to handle the largest expected input for the item.

GetObjectDefMaxArgs

Extensible input techniques

While developers do their best to guess how many items are needed in an object, users will often want to extend that object with far more fields than were dreamed of. Using Allocatable arrays in Fortran usually makes this feasible, the special \extensible field makes it possible.

Example:

```
USE InputProcessor, ONLY: GetObjectDefMaxArgs
---
CHARACTER(len=MaxNameLength), ALLOCATABLE, DIMENSION(:) :: Alphas
REAL, ALLOCATABLE, DIMENSION(:) :: Numbers

! You supply the object word, routine returns numargs, numalpha, numnumeric
CALL GetObjectDefMaxArgs('DAYSCHEDULE:INTERVAL',NumArgs,NumAlpha,NumNumeric)

ALLOCATE(Alphas(NumAlpha))
ALLOCATE(Numbers(NumNumeric))

! Then, usual get calls...
```

Thus, you can determine how many arguments that the IDD has defined as “maximum” for a given object.

GetObjectItemNum

GetObjectItem, described above, requires the input file index of the desired object in order to get the object's data. Sometimes this index may be unknown, but the name of the object is known. GetObjectItemNum returns the input file index given the class name and object name.

```
Example:
USE InputProcessor, ONLY: GetObjectItemNum
---
ListNum = GetObjectItemNum('CONTROLLER LIST',ControllerListName)
```

In the example, ListNum will contain the input file index of the 'CONTROLLER LIST' whose name is contained in the string variable ControllerListName.

FindItemInList

This function looks up a string in a similar list of items and returns the index of the item in the list, if found. It is case sensitive.

```
Example:
USE InputProcessor, ONLY: FindItemInlist
---
SysNum = FindItemInList(CompName,Sys%SysName,NumSys)
```

CompName is the input string, Sys%SysName is the list of names to be searched, and NumSys is the size of the list.

A case insensitive version of the routine is called FindItem (same description).

SameString

This function returns true if two strings are equal (case insensitively).

```
Example:
USE InputProcessor, ONLY: SameString
---
IF (SameString(InputRoughness, 'VeryRough')) THEN
    Material(MaterNum)%Roughness=VeryRough
ENDIF
```

VerifyName

This subroutine checks that an object name is unique; that is, it hasn't already been used for the same class of object and the name is not blank.

```
Example:
USE InputProcessor, ONLY: VerifyName
---
CALL VerifyName(AlphaArray(1), Fan%FanName, &
    FanNum-1, IsNotOK, IsBlank, 'FAN:SIMPLE:CONSTVOLUME Name')
```

The first argument is the name to be checked, the second is the list of names to search, the third argument is the number of entries in the list, the 4th argument is set to TRUE if verification fails, the 5th argument is set to true if the name is blank, and the last argument is part of the error message written to the error file when verification fails.

RangeCheck

The routine RangeCheck can be used to produce a reasonable error message to describe the situation in addition to setting the ErrorsFound variable to true. Errors found can then be checked in the calling routine and the program terminated if desired.

```
SUBROUTINE RangeCheck(ErrorsFound, WhatFieldString, WhatObjectString, ErrorLevel, &
    LowerBoundString, LowerBoundCondition, UpperBoundString, UpperBoundCondition)
```

It can be used in a variety of places when the \minimum and \maximum fields will not work (e.g. different min/max dependent on some other field).

```
USE InputProcessor, ONLY: RangeCheck
---
ErrorsFound=.false.
CALL RangeCheck(ErrorsFound, 'DryBulb Temperature', 'WeatherFile', &
    'SEVERE', '> -70', (Drybulb>-70.), '< 70', (DryBulb <70.))
CALL RangeCheck(ErrorsFound, 'DewPoint Temperature', 'WeatherFile', &
    'SEVERE', '> -70', (Dewpoint>-70.), '< 70', (Dewpoint <70.))
CALL RangeCheck(ErrorsFound, 'Relative Humidity', 'WeatherFile', &
    'SEVERE', '> 0', (RelHum>=0.), '<= 110', (RelHum<=110.))
```

To examine one call:

The variable **DryBulb** is set to its value. In this case, it is coming from the **Weather File**. The **LowerBoundString** is '> -70' and the **LowerBoundCondition** is (DryBulb>-70.) [this expression will yield true or false depending...]

The LowerBounds (**LowerBoundString**, **LowerBoundCondition**) are optional as are the UpperBounds (**UpperBoundString**, **UpperBoundCondition**). If we were only testing one set of ranges, the call would look like:

```
Call RangeCheck(ErrorsFound,'DryBulb Temperature','WeatherFile','SEVERE', &
  UpperBoundString='< 70', UpperBoundCondition=(DryBulb<70.))
```

ErrorLevel can be one of the usual Error levels:

WARNING – would be a simple warning message – the calling routine might reset the value to be within bounds

SEVERE – a severe error. Usually the program would terminate if this is in a “GetInput” routine. If during execution, the calling program could reset the value but RangeCheck contains too many string comparisons to be called for an execution problem.

FATAL – not likely to be used. You want to provide a context to the error and if really a fatal type error, you'd like to execute the RangeCheck call and then terminate from the calling program.

And the context for the message may be shown in the calling routine by checking the value of ErrorsFound:

```
ErrFound=.false.
Call RangeCheck(ErrFound,'This field','SEVERE','<= 100',(Value<100.))
IF (ErrFound) THEN
  CALL ShowContinueError('Occurs in routine xyz')
  ErrorsFound=.true. ! for later termination
ENDIF
```

MakeUPPERCase

This function can be used to make sure an upper case string is being used. (Note this is not needed when using “SameString”). Parameter 1 to the function is the string to be upper cased:

```
USE InputProcessor, ONLY: MakeUPPERCase
---
UCString=MakeUPPERCase('lower string')
```

NodeInputManager

The NodeInputManager is responsible for getting all the node names and assigning each a number. Node names are learned in random order – which can make validation difficult. Internally nodes are referenced as number and should be integers in any data structure or reference. Two key routines are used for obtaining node numbers: GetOnlySingleNode and GetNodeNums. Both routines need some extra information about the node as the number is obtained:

NodeFluidType – Proper definitions for this can be found in DataLoopNode.

NodeObjectType – This is the object for the node (e.g. Chiller:Electric)

NodeObjectName – this is the object's name field (e.g. My Chiller)

NodeConnectionType – Again, proper definitions can be found in DataLoopNode

NodeFluidStream – which fluid stream this belongs to (1,2,3)

ObjectIsParent – True If the object is a parent object, false if not

GetOnlySingleNode

This is used when only one node is expected as the input point. If this name points to a NodeList, an appropriate error message will be issued and errFlag (the second argument) will be set .true.

GetOnlySingleNode(NodeName,errFlag,NodeObjectType,NodeObjectName,NodeFluidType,NodeConnectionType,NodeFluidStream,ObjectIsParent)

It is used:

```
Example:
USE NodeInputManager, ONLY: GetOnlySingleNode
. . .
! get inlet node number
Baseboard(BaseboardNum)%WaterInletNode = &
    GetOnlySingleNode(AlphaArray(3),ErrorsFound, &
        'Baseboard Heater:Water:Convective',AlphaArray(1), &
        NodeType_Water,NodeConnectionType_Inlet, &
        1,ObjectIsNotParent)
! get outlet node number
Baseboard(BaseboardNum)%WaterOutletNode = &
    GetOnlySingleNode(AlphaArray(4),ErrorsFound, &
        'Baseboard Heater:Water:Convective',AlphaArray(1), &
        NodeType_Water,NodeConnectionType_Outlet, &
        1,ObjectIsNotParent)
```

The first argument is the node name, the 2nd argument is the error flag variable, the 3rd argument is the object type, the 4th argument is the object name – the remainder arguments are as listed above.

GetNodeNums

This is used when more than one node is valid for an input. Like the GetOnlySingleNode invocation, GetNodeNums needs the extra information for a node:

```
SUBROUTINE GetNodeNums(Name,NumNodes,NodeNumbers,ErrorsFound, &
    NodeFluidType,NodeObjectType,NodeObjectName, &
    NodeConnectionType,NodeFluidStream,ObjectIsParent)

Example:
USE NodeInputManager, ONLY: GetNodeNums
. . .
CHARACTER(len=MaxNameLength), DIMENSION(4) :: AlphaArray
INTEGER :: NumNodes
INTEGER, DIMENSION(25) :: NodeNums
. . .
! Get the supply nodes
ErrInList=.false.
CALL GetNodeNums(Names(8),NumNodes,NodeNums,ErrInList,NodeType_Air, &
    'AIR PRIMARY LOOP',PrimaryAirSystem(AirSysNum)%Name, &
    NodeConnectionType_Inlet,1,ObjectIsParent)
IF (ErrInList) THEN
    CALL ShowContinueError('Invalid Node Name or Node List in Air System=' &
        //TRIM(PrimaryAirSystem(AirSysNum)%Name))
    ErrorsFound=.true.
ENDIF
! Allow at most 3 supply nodes (for a 3 deck system)
IF (NumNodes > 3) THEN
    CALL ShowSevereError('Air System:Only 1st 3 Nodes will be used from:' &
        //TRIM(Names(8)))
    CALL ShowContinueError('Occurs in Air System='// &
        TRIM(PrimaryAirSystem(AirSysNum)%Name))
    ErrorsFound=.true.
ENDIF
```

```

IF (NumNodes.EQ.0) THEN
  CALL ShowSevereError('Air System:there must be at least 1 '// &
    'supply node in system '//TRIM(NAMES(1)))
  CALL ShowContinueError('Occurs in Air System='// &
    TRIM(PrimaryAirSystem(AirSysNum)%Name))
  ErrorsFound=.true.
END IF
. . . . .

```

The first argument is a node name or the name of a Node List, the 2nd argument is the number of nodes in the Node List (1 for a single node), the 3rd argument is the output: a list of node numbers – these are followed by the arguments shown above.

Unique Node Checking

A set of routines will allow you to use the NodeInputManager to check for unique node names across a set of inputs. This is used currently in the CONTROLLED ZONE EQUIP CONFIGURATION object where each zone node mentioned must be unique. Three routines comprise the unique node check: InitUniqueNodeCheck, CheckUniqueNodes, EndUniqueNodeCheck

InitUniqueNodeCheck

A call to this routine starts the collection and detection of unique/non-unique nodes by the NodeInputManager:

```

USE NodeInputManager, ONLY: InitUniqueNodeCheck, CheckUniqueNodes, &
  EndUniqueNodeCheck
. . .
CALL InitUniqueNodeCheck('CONTROLLED ZONE EQUIP CONFIGURATION')

```

The only argument is a simple string that will help with error messages that may come from the NodeInputManager. Unique node checking can only be done for one context ('CONTROLLED ZONE EQUIP CONFIGURATION') at a time.

CheckUniqueNodes

<pre> SUBROUTINE CheckUniqueNodes(NodeTypes,CheckType,ErrorsFound, & CheckName,CheckNumber) </pre>
--

This is the routine called during the getting of the nodes. The CheckType argument can be 'NodeName' or 'NodeNumber' and then pass in the appropriate argument to CheckName or CheckNumber. CheckName and CheckNumber are optional arguments – only the necessary one need be supplied.

Argument 1, NodeTypes, is the type of node being looked for – this argument is used for error messages within the NodeInput processing. Argument 2, ErrorsFound, will be set to true if this node is not unique in the current context.

```

Example:
  UniqueNodeError=.false.
  CALL CheckUniqueNodes('Zone Air Node','NodeName',UniqueNodeError, &
    CheckName=AlphaArray(5))
  IF (UniqueNodeError) THEN
    CALL ShowContinueError('Occurs for Zone='//TRIM(AlphaArray(1)))
    ErrorsFound=.true.
  ENDIF

```

EndUniqueNodeCheck

This routine terminates the unique node check – allows arrays to be deallocated, etc.

```
CALL EndUniqueNodeCheck('CONTROLLED_ZONE_EQUIP_CONFIGURATION')
```

The only argument is the Context String – which must match the string given in the InitUniqueNodeCheck routine.

SetUpCompSets and TestCompSet

SetUpCompSets and TestCompSet are used to develop a list of hierarchical relationships between HVAC objects. A list of component sets is built which contains the following information:

Parent Object Type (Currently cannot be SPLITTER or MIXER)

Parent Object Name

Child Component Type (Currently cannot be SPLITTER or MIXER)

Child Component Name

Child Component InletNodeName

Child Component OutletNodeName

Node Description

Parent and child refer to a hierarchical relationship of two HVAC objects. For example, a branch is the parent to a pump, and a fan coil is the parent to a fan. The component sets do not include peer-to-peer connections such as a splitter connected to a branch, or a zone supply air path connected to an air loop.

The following rules apply to component sets:

- Each parent/child component set is unique. The same pair of components should never appear in the component sets list more than once.
- Each set of child component plus inlet and outlet nodes is unique.
- Each child component must have a parent component.
- A given component may appear in multiple component sets as a parent component.
- A given component may appear in multiple component sets as a child component only if there is a different (Incomplete sentence here).
- If a given node name appears more than once as an inlet node, the two components which use it must share a parent/child relationship.
- If a given node name appears more than once as an outlet node, the two components which use it must share a parent/child relationship.
- After all input data have been read in by the program, there should be no "UNDEFINED" values in the list of component sets.

When any of these rules are violated, a warning is issued indicating a possible node connection error.

```
** Warning ** Potential Node Connection Error for object PIPE,
name=CW_BYPASS
**   ~~~   **   Node Types are still UNDEFINED -- See Branch/Node
Details file for further information
**   ~~~   **   Inlet Node : CW_BYPASS_INLET
**   ~~~   **   Outlet Node: CW_BYPASS_OUTLET
```

The component sets are reported in the eplusout.bnd file:

```
! <Component Set>,<Component Set Count>,<Parent Object Type>,<Parent
Object Name>,<Component Type>,<Component Name>,<Inlet Node ID>,<Outlet
Node ID>,<Description>

Component Set,1,BRANCH,COOLING SUPPLY INLET BRANCH,PUMP:VARIABLE
SPEED,CHW CIRC PUMP,CHW SUPPLY INLET NODE,CHW PUMP OUTLET NODE,Water
Nodes

Component Set,21,FAN COIL UNIT:4
PIPE,ZONE1FANCOIL,FAN:SIMPLE:CONSTVOLUME,ZONE1FANCOILFAN,ZONE1FANCOILOAMI
XEROUTLETNODE,ZONE1FANCOILFANOUTLETNODE,Air Nodes
```

SetUpCompSets

SetUpCompSets should be called any time a parent object such as a branch or a compound object (e.g. furnace) references a child component which is connected to it. If an object has more than one child component, then SetUpCompSets is called once for each child.

SetUpCompSets first looks for the child component in the existing list of component sets by looking for a matching component type and name. If it is found, then the parent name and type are filled in. If the child component is not found in the existing list, then a new component set is created.

```
SUBROUTINE
  SetUpCompSets(ParentType,ParentName,CompType,CompName,InletNode,Outlet
Node,Description)
```

The arguments are:

ParentType	Parent Object Type
ParentName	Parent Object Name
CompType	Child Component Type
CompName	Child Component Name
InletNode	Child Component Inlet Node Name
OutletNode	Child Component Outlet Node Name
Description	Description of nodes (optional)

For example, FURNACE:BLOWTHRU:HEATONLY references a fan and a heating coil:
The latest version of Furnace:blowthru:heatonly has min-fields 16 Update idd object below to reflect latest...

```
FURNACE:BLOWTHRU:HEATONLY,
  \min-fields 15
  A1,  \field Name of furnace
  A2,  \field Availability schedule
  A3,  \field Furnace inlet node name
  A4,  \field Furnace outlet node name
  A5,  \field Operating Mode
  N1,  \field Furnace heating capacity
  N2,  \field Maximum supply air temperature from furnace heater
  N3,  \field Design air flow rate through the furnace
  A6,  \field Controlling zone or thermostat location
  N4,  \field Fraction of the total or design volume flow that goes
through the controlling zone
  A7,  \field Supply fan type
  A8,  \field Supply fan name
  A9,  \field Fan outlet node
  A10, \field Heating coil type
  A11; \field Heating coil name
```

In this case, the furnace is the parent object to the fan and the heating coil. To set up the component set for the furnace and its fan, the furnace type and name, the fan type and name (A7 and A8), and the fan inlet and outlet nodes (A3 and A9) are passed to SetupCompSets:

Example:

```
CALL SetupCompSets(
  Furnace(FurnaceNum)%FurnaceType, Furnace(FurnaceNum)%Name,
  AlphArray(7), AlphArray(8), AlphArray(3), AlphArray(9))
```

In some cases, the inlet or outlet node names may not be known by the parent object. In this case, "UNDEFINED" is passed to SetupCompSets.

Examples:

```
CALL SetupCompSets(
  Furnace(FurnaceNum)%FurnaceType, Furnace(FurnaceNum)%Name,
  AlphArray(8), AlphArray(9), 'UNDEFINED', 'UNDEFINED')

CALL SetupCompSets(
  Furnace(FurnaceNum)%FurnaceType, Furnace(FurnaceNum)%Name,
  AlphArray(12), AlphArray(13), 'UNDEFINED', AlphArray(4))
```

TestCompSet

The NOTE below doesn't make sense to me. Are you going to update the text in this section to reflect the new purpose for this subroutine at some point? You might want to delete this section until you have a chance to update this text, to avoid confusion.... Or say NOTE: The comments for TestCompSet (below) are old. They reflect a function as originally designed, but it's a subroutine which has a different purpose now. The text in this section will be updated in the near future to reflect this different purpose.

NOTE: The comments in testcompset are old. They describe a function as originally designed, but it's a subroutine which has a different purpose now.

TestCompSet should be called by every HVAC object which has a parent object. A given object may be both a parent and a child. For example, FURNACE:BLOWTHRU:HEATONLY is a child to a branch and a parent to a fan and coils.

TestCompSet first looks for the calling component in the existing list of component sets by looking for a matching component type and name. If it is found, then any undefined node names are filled in and the description string for the nodes is added. If the component is not found, then a new component set is created with undefined parent object type and name.

```
SUBROUTINE
TestCompSet (CompType, CompName, InletNode, OutletNode, Description)
```

The arguments are:

CompType	Child Component Type
CompName	Child Component Name
InletNode	Child Component Inlet Node Name
OutletNode	Child Component Outlet Node Name
Description	Description of nodes

For example, FURNACE:BLOWTHRU:HEATONLY is a child component with inlet and outlet nodes:

```
FURNACE:BLOWTHRU:HEATONLY,
  \min-fields 15
  A1,  \field Name of furnace
  A2,  \field Availability schedule
  A3,  \field Furnace inlet node name
  A4,  \field Furnace outlet node name
```

To check the component set for the furnace, the furnace type and name, and the furnace inlet and outlet nodes (A3 and A4) along with a node descriptor are passed to TestCompSets:

Example:

```
CALL TestCompSet (Furnace(FurnaceNum)%FurnaceType, AlphArray(1)
, AlphArray(3), AlphArray(4), 'Air Nodes')
```

Schedule Services

Schedules are widely used in specifying input for building simulation programs. For instance heat gains from lighting, equipment and occupancy are usually specified using schedules. Schedules are used to indicate when equipment is on or off. Schedules are

also used to specify zone and system set points. EnergyPlus uses schedules in all these ways and provides services that make using schedules very easy for the developer.

Schedules are specified in a three level hierarchy in EnergyPlus input.

DaySchedules (IDD Objects: DaySchedule, DaySchedule:Interval, DaySchedule:List)

WeekSchedules (IDD Objects: WeekSchedule, WeekSchedule:Compact)

Schedules (IDD Objects: Schedule, Schedule:Compact)

In addition, a **ScheduleType** object can specify certain limits on the schedules. This is a mostly optional input but can be used effectively. (That is, if your examples include it, users will probably use it too.)

An example from an input (IDF) file:

```
ScheduleType,
  Fraction,  !- ScheduleType Name
  0.0 : 1.0,  !- range
  CONTINUOUS;  !- Numeric Type

ScheduleType,
  On/Off,  !- ScheduleType Name
  0:1,  !- range
  DISCRETE;  !- Numeric Type

! Schedule Constant
SCHEDULE:COMPACT,
Constant,
on/off,
Through: 12/31,
For: AllDays,
Until: 24:00, 1.0;
```

```
! Schedule Daytime Ventilation
SCHEDULE:COMPACT,
Daytime Ventilation,
Fraction,
Through: 12/31,
For: Weekdays SummerDesignDay,
Until: 08:00, 0.0,
Until: 18:00, 1.0,
Until: 24:00, 0.0,
For: Weekends WinterDesignDay,
Until: 10:00, 0.0,
Until: 16:00, 1.0,
Until: 24:00, 0.0,
For: Holidays AllOtherDays,
Until: 24:00, 0.0;

! Schedule Intermittent
SCHEDULE:COMPACT,
Intermittent,
Fraction,
Through: 12/31,
For: AllDays,
Until: 08:00, 0.0,
Until: 18:00, 1.0,
Until: 24:00, 0.0;
```

The day schedule elements assign numbers that span a full day (24 hours). The week schedule elements indicates which day schedules are applicable to each day of the

week plus holiday and some special days. Schedule elements indicate which week schedules are applicable to various periods of the year. Both day schedules and schedules reference a schedule type. A schedule type is characterized by a range (e.g. 0 to 1) and whether it is continuous (can assume any value) or discrete (can assume integer values only). The following routines from the ScheduleManager module enable the developer to use schedules in a simulation.

GetScheduleIndex

This function takes a schedule name as input and returns an internal pointer to the schedule. Schedule values will always be accessed via the pointer not the name during the simulation for reasons of efficiency. This function should be called once for each schedule during the input phase and the returned value stored in the appropriate data structure.

```
Example:
USE ScheduleManager, ONLY: GetScheduleIndex
. . .
Baseboard(BaseboardNum)%SchedPtr = GetScheduleIndex(AlphArray(2))
```

Here the schedule pointer for the schedule name contained in AlphArray(2) is stored in the baseboard data structure for later use. If a 0 is returned, this is not a valid schedule. Objects should also typically check for “blank” schedules.

CheckScheduleValueMinMax

Since you can't always rely on a user to input the ScheduleType, the ScheduleManager can be used to check the minimum and/or maximum values for a schedule.

<pre>LOGICAL FUNCTION CheckScheduleValueMinMax(ScheduleIndex, & MinString, Minimum, MaxString, Maximum)</pre>
--

The pair of specifications (MinString, Minimum) and (MaxString, Maximum) are optional -- only one set need be given.

Examples from the code:

```
USE ScheduleManager, ONLY: CheckScheduleValueMinMax
. . .
IF (.NOT. CheckScheduleValueMinMax(ScheduleIndex, '>=', 0., '<=', 1.)) THEN
CALL ShowSevereError('SET POINT MANAGER:SINGLE ZONE MIN HUM, humidity..')
CALL ShowContinueError('Error found in schedule = '//TRIM(Alphas(3)))
CALL ShowContinueError('set point values must be (>=0., <=1.)')
ErrorsFound=.true.
END IF
```


GetScheduleMinValue

There are times when you don't necessarily want to issue an error message but might like to find out what the minimum value of a given schedule is. For example, if the schedule allowed for >1 multipliers on a given input.

<pre>FUNCTION GetScheduleMinValue(ScheduleIndex) RESULT(MinimumValue)</pre>

Example of use:

```
USE ScheduleManager, ONLY: GetScheduleMinValue
. . .
Value=GetScheduleMinValue(ScheduleIndex)
```

The only argument needed is the ScheduleIndex for the schedule. The return value is real –  cannot differentiate integer values[DBS1].

GetScheduleMaxValue

There are times when you don't necessarily want to issue an error message but might like to find out what the maximum value of a given schedule is. For example, if the schedule allowed for >1 multipliers on a given input.

```
FUNCTION GetScheduleMaxValue(ScheduleIndex) RESULT(MaximumValue)
```

Example of use:

```
USE ScheduleManager, ONLY: GetScheduleMaxValue
...
Value=GetScheduleMaxValue(ScheduleIndex)
```

The only argument needed is the ScheduleIndex for the schedule. The return value is real – it cannot differentiate integer values.

GetCurrentScheduleValue

This function returns the current schedule value for the current day and time, given the schedule pointer as input. An optional second argument can be used to specify a different hour than the current global hour of the day.

```
Example
USE ScheduleManager, ONLY: GetCurrentScheduleValue
...
CloUnit = GetCurrentScheduleValue(People(PeopleNum)%ClothingPtr)
```

Notice that the developer doesn't have to keep track of hour of the day, day of the month, or month. The program does all of that. The only input needed is the pointer to the schedule. Should you include an example for the "optional second argument?"

Other Useful Utilities

GetNewUnitNumber

Rather than attempt to keep track of all open files and distribute this list to everyone, we have chosen to use a routine that does this operation. If you need to have a scratch file (perhaps when porting legacy code into EnergyPlus modules), you can use the GetNewUnitNumber function to determine a logical file number for the OPEN and READ/WRITE commands. The function works by looking at all open assigned files and returning a number that isn't being used. This implies that you will OPEN the unit immediately after calling the function (and you should!).

```
Example:
INTEGER, EXTERNAL :: GetNewUnitNumber
...
myunit=GetNewUnitNumber()
OPEN(Unit=myunit,File='myscratch')
```

FindUnitNumber

If you want to find out a unit number for a file you think is already open, you can use the FindUnitNumber function. For example, rather than creating a new unit for debug output, you could latch onto the same unit as currently used for the "eplusout.dbg" file.

```
Example:
INTEGER, EXTERNAL :: FindUnitNumber
...
myunit=FindUnitNumber('eplusout.dbg')
```

If that file is already opened, it will get back the unit number it is currently assigned to. If it is not opened or does not exist, it will go ahead, get a unit number, and OPEN the file. (Should not be used for Direct Access or Binary files!)

FindNumberinList

Sometimes you would like to find a number in a list. This is applicable to integers only (e.g. Index numbers of some item).

```
Example:
INTEGER, EXTERNAL :: FindUnitNumber
...
MatchingCooledZoneNum = &
    FindNumberinList(CtrlZoneNum, &
        AirToZoneNodeInfo(AirLoopNum)%CoolCtrlZoneNums, NumZonesCooled)
```

The location/index in the array `AirToZoneNodeInfo%CoolCtrlZoneNums` will be returned if it finds the number in the array. If 0 is returned, it did not find that number in the list.

ValidateComponent

Many objects specify a component type as well as a component name. Or, an object might have only a component name. The `ValidateComponent` routine will allow for objects outside the scope of a current "GetInput" routine to verify that the specific component does exist in the input file.

```
SUBROUTINE ValidateComponent(CompType, CompName, IsNotOK, CallString)
```

`CompType`, `CompName` are the typical nomenclature for "Component Type" (e.g. Fan:Simple:OnOff) and "Component Name" (e.g. "my fan" – user specified). `IsNotOk` is a logical from the calling program that is set to true when the component is not on the input file. `CallString` should specify the calling object – so that an appropriate error message can be issued.

```
Example:
! No USE needed - straightforward routine in GeneralRoutines
CALL ValidateComponent(FurnaceNum)%FanType, &
    Furnace(FurnaceNum)%FanName, IsNotOK, &
    'Furnace:BlowThru:HeatOnly')
IF (IsNotOK) THEN
    CALL ShowContinueError('In Furnace='// &
        TRIM(Furnace(FurnaceNum)%Name))
    ErrorsFound=.true.
ENDIF
```

Note that in the example, the `FanType` is entered by the user. This allows for ultimate flexibility though the example could also include appropriate fan types that are inherent to the code (an acceptable, if somewhat inflexible, practice).

CheckComponent

This routine is exactly like `ValidateComponent` but doesn't generate an error message. It could be used instead of `ValidateComponent` and you could use the "IsNoOK" to generate your own error message. However, the intended use is for checking out different components when you don't have the component type as a field for the object. Thus, you can easily check if there is an object (component type) with the name entered in your field.

```
SUBROUTINE CheckComponent(CompType, CompName, IsNotOK)
```

CompType, CompName are the typical nomenclature for “Component Type” (e.g. Fan:Simple:OnOff) and “Component Name” (e.g. “my fan” – user specified). IsNotOk is a logical from the calling program that is set to true when the component is not on the input file.

```
Example:
! No USE needed - straightforward routine in GeneralRoutines
CALL CheckComponent ('Furnace:BlowThru:HeatOnly', &
    FurnaceRefName, IsNotOK)
IF (IsNotOK) THEN
    CALL CheckComponent ('Furnace:BlowThru:HeatCool', &
        FurnaceRefName, IsNotOK)
    . . . more checks on IsNotOK
ELSE
    FurnaceType= 'Furnace:BlowThru:HeatOnly'
ENDIF
. . .
```

Note that in the example, the FurnaceRefName is entered by the user. And this module knows what kind of components it might be.

CreateSysTimeIntervalString

A very important part of EnergyPlus simulation is to be able to alert the user to problems during the simulation. The CreateSysTimeIntervalString will help do that though a better use is the ShowContinueErrorTimeStamp routine. The routine has no argument – a string is returned. The example below also illustrates the preferred method of counting how many times an error is produced and not printing each occurrence.

```
Example:
USE General, ONLY: CreateSysTimeInterval
---
!The warning message will be suppressed during the warm up days.
If (.NOT.WarmUpFlag) Then
    ErrCount = ErrCount + 1
    IF (ErrCount < 15) THEN
        CALL ShowWarningError('SimAirLoops: Max iterations exceeded for '// &
            TRIM(PrimaryAirSystem(AirLoopNum)%Name)//', at '// &
            TRIM(EnvironmentName)//', '//TRIM(CurMnDy)//' '// &
            TRIM(CreateSysTimeIntervalString()))
    ELSE
        IF (MOD(ErrCount,50) == 0) THEN
            WRITE(CharErrOut,*) ErrCount
            CharErrOut=ADJUSTL(CharErrOut)
            CALL ShowWarningError ('SimAirLoops: Exceeding max iterations'// &
                ' continues...'//CharErrOut)
        ENDIF
    ENDIF
End If
```

TrimSigDigits

Along with error messages to alert the user, oftentimes you'd like to include values that are in error. You can use what some of the examples have shown – Write(string,*) value but that will produce many digits in real numbers. The TrimSigDigits routine will allow for easy modification to a set of digits.

```
FUNCTION TrimSigDigits(RealValue,SigDigits) RESULT(OutputString)
```

As seen in the following example of use in code, a real value is passed in as argument 1 and the number of digits desired is passed in as argument 2. Note that the routine will preserve any “E+xx” outputs when a value like .000000004 might be passed in.

```

USE General, ONLY: TrimSigDigits
. . .
CALL ShowWarningError('COIL:Water:DetailedFlatCooling in Coil ='// &
                     TRIM(WaterCoil(coilNum)%Name))
CALL ShowContinueError('Air Flow Rate Velocity has greatly exceeded '// &
                     'upper design guidelines of ~2.5 m/s')
CALL ShowContinueError('Air MassFlowRate[kg/s]='// &
                     TRIM(TrimSigDigits(AirMassFlow,6)))

AirVelocity=AirMassFlow*AirDensity/WaterCoil(CoilNum)%MinAirFlowArea
CALL ShowContinueError('Air Face Velocity[m/s]='// &
                     TRIM(TrimSigDigits(AirVelocity,6)))
CALL ShowContinueError('Approximate MassFlowRate limit for Face '// &
                     Area[kg/s]='// &
                     TRIM(TrimSigDigits(2.5*WaterCoil(CoilNum)%MinAirFlowArea/AirDensity,6)))
CALL ShowContinueError('COIL:Water:DetailedFlatCooling could be '// &
                     'resized/autosized to handle capacity')
CoilWarningOnceFlag(CoilNum) = .False.

```

RoundSigDigits

Similar to TrimSigDigits, the RoundSigDigits function may be used when you want to “round” the output string – perhaps for reporting and/or error messages.

```
FUNCTION RoundSigDigits(RealValue,SigDigits) RESULT(OutputString)
```

As seen in the following example of use in code, a real value is passed in as argument 1 and the number of digits desired is passed in as argument 2. Note that the routine will preserve any “E+xx” outputs when a value like .000000004 might be passed in.

```

USE General, ONLY: RoundSigDigits
. . .
LatOut=RoundSigDigits(Latitude,2)
LongOut=RoundSigDigits(Longitude,2)
TZOut=RoundSigDigits(TimeZoneNumber,2)
NumOut=RoundSigDigits(Elevation,2)
PressOut=RoundSigDigits(StdBaroPress,0)
Write(OutputFileInits,LocFormat) Trim(LocationTitle),TRIM(LatOut), &
                                TRIM(LongOut), &
                                TRIM(TZOut), &
                                TRIM(NumOut), &
                                TRIM(PressOut)

```

Error Messages

Three error message routines are provided for the developer, indicating three different levels of error severity: ShowFatalError, ShowSevereError, and ShowWarningError. Each takes a string as an argument. The string is printed out as the message body on the file “eplusout.err”. There are two additional optional arguments, which are file unit numbers on which the message will also be printed. ShowFatalError causes the program to immediately abort.

Two other error messages can be used to help make the error file more readable: ShowContinueError and ShowContinueErrorTimeStamp. Finally, another similar ShowMessage call can be used to display an informative string to the error file (eplusout.err).

As indicated, all of the “show” error calls look the same:


```
SUBROUTINE <ErrorMessageCall>(ErrorMessage,OutUnit1,OutUnit2)
```

Or

```
SUBROUTINE ShowWarningError(ErrorMessage,OutUnit1,OutUnit2)
SUBROUTINE ShowSevereError(ErrorMessage,OutUnit1,OutUnit2)
SUBROUTINE ShowFatalError(ErrorMessage,OutUnit1,OutUnit2)
SUBROUTINE ShowContinueError(ErrorMessage,OutUnit1,OutUnit2)
SUBROUTINE ShowContinueErrorTimeStamp(ErrorMessage,OutUnit1,OutUnit2)
SUBROUTINE ShowMessage(Message,OutUnit1,OutUnit2)
```

Mostly, you would never use either of the optional “OutUnit” arguments. One use might be if you were, in addition to the normal EnergyPlus output files, writing your own output file that would be processed separately.

Due to the optional parameters, Interface statements are set in DataGlobals and you must enter USE statements defining which of the error calls you wish to use.

Example:

```
USE DataGlobals, ONLY: ShowSevereError
. . .
IF (Construct(ConstrNum)%LayerPoint(Layer) == 0) THEN
  CALL ShowSevereError('Did not find matching material for construct ` &
    //TRIM(Construct(ConstrNum)%Name)// ` &
    `, missing material = ` &
    //TRIM(ConstructAlphas(Layer))`
  ErrorsFound=.true.
ENDIF
```

This code segment will produce (with proper conditions) the message onto the error file:

```
** Warning ** Did not find matching material for construct XYZ, missing
material = ABC
```

The ShowContinueError is used in conjunction with either ShowSevereError or ShowWarningError. The “~~~” characters represent the continuation:

```
** Warning ** The total number of floors, walls, roofs and internal mass
surfaces in Zone ZONE ONE
** ~~~ ** is < 6. This may cause an inaccurate zone heat balance
calculation.
** Warning ** No floor exists in Zone=ZONE ONE
** Warning ** Surfaces in Zone="ZONE ONE" do not define an enclosure.
** ~~~ ** Number of surfaces is <= 4 in this zone. View factor
reciprocity forced
```

The ShowContinueError is particularly useful with some of the previous routines that, in addition to signaling an error, produce their own error message. For example, see the example code in the ValidateComponent excerpt above. Note that no ShowContinueError should be used with the ShowFatalError as it immediately terminates the program. Instead, a Severe-Continue-Fatal sequence should be used.

Each GetInput routine is responsible for verifying its input. Rather than terminating with the first illegal value, however, it is better to have an “ErrorsFound” logical that gets set to true for error conditions during the main routine processing and terminates at the end of the GetInput routine. Of course during simulation, conditions should also be checked and terminated if necessary. Try to give the user as much information as possible with the set of error routine calls.

Quite a complex message can be constructed using concatenation. These routines can also be used to output numeric fields by writing the numeric variables to a string variable, although this isn’t very convenient.

A good use of the ContinueErrorTimeStamp as well as “counting” errors is shown below:

```

IF(OutDryBulbTemp .LT. 0.0) THEN
  CINErrCount1=CINErrCount1+1
  IF (CINErrCount1 < 15) THEN
    CALL ShowWarningError('ElectricChillerModel:Air Cooled '// &
      'Condenser Inlet Temperature below 0C')
    CALL ShowContinueErrorTimeStamp('OutDoor Dry Bulb='// &
      TRIM(RoundSigDigits(OutDryBulbTemp,2)('//',''))
  ELSE
    IF (MOD(CINErrCount1,50) == 0) THEN
      WRITE(CINCharErrOut,*) CINErrCount1
      CINCharErrOut=ADJUSTL(CINCharErrOut)
      CALL ShowWarningError('ElectricChillerModel:Air Cooled'// &
        ' Condenser Inlet Temperature below 0C continues...' &
        //CINCharErrOut)
    ENDIF
  ENDIF
ENDIF

```

Display Strings

Two display routines are useful for displaying to the “run” log the progress of the simulation. Since EnergyPlus usually runs as a “console” mode application, users may monitor progress of the simulation. Thus, at times it is useful to have messages displayed there. These should be minimal in number though can be used effectively during debugging of new modules.

```

subroutine DisplayString(String)
subroutine DisplayNumberandString(Number,String)

```

The “String” parameters are normal strings. The “Number” parameter must be an integer.

Performance Curve Services

Some HVAC equipment models in EnergyPlus use performance curves. These are polynomials in one or two independent variables that are used to modify rated equipment performance for performance at the current, off-rated conditions. Most often the curves are functions of temperature – entering wetbulb and outside drybulb, for instance – or of the part load fraction. EnergyPlus provides services to input, store, and retrieve curve data and to evaluate curves given values of the independent variables. There are 3 curve objects: CURVE:QUADRATIC, CURVE:CUBIC, and CURVE:BIQUADRATIC.

GetCurveIndex

This function takes a curve name as input and returns an internal pointer to the curve. Curve values will always be accessed via the pointer not the name during the simulation for reasons of efficiency. This function is usually called once for each curve during the input phase.

```

USE CurveManage, ONLY: GetCurveIndex
...
DXCoil(DXCoilNum)%CCapFTemp = GetCurveIndex(Alphas(5))
IF (DXCoil(DXCoilNum)%CCapFTemp .EQ. 0) THEN
  CALL ShowSevereError('COIL:DX:BF-Empirical not found=' &
    //TRIM(Alphas(5)))
  ErrorsFound = .TRUE.
END IF

```

CurveValue

This function takes the curves index and one or two independent variables as input and returns the curve value.

```
USE CurveManage, ONLY: CurveValue
. . .
! Get total capacity modifying factor (function of temperature)
! for off-rated conditions
50 TotCapTempModFac = CurveValue(DXCoil(DXCoilNum)%CCapFTemp,
                                InletAirWetbulbC, &
                                OutDryBulbTemp)
```

Fluid Property Services

Fluid property routines have been implemented within EnergyPlus with the goal of making the specification of new fluids relatively easy for the user and (starting with version 1.2.1) not require the user to specify data for the most common loop fluids. Common refrigerants are listed within an extensive Reference Data Set (RDS) that is provided with the EnergyPlus program.

Fluids in EnergyPlus are broken into two categories: refrigerants and glycols. This relates back to the amount of information needed to determine the properties of the various fluid types inside the program. The decision to define or use one class of fluids or another relates back to whether or not one expects the fluid to change phase (liquid and/or vapor) or remain a liquid. When a developer feels that a fluid may change phase, all code should access the Refrigerant class of fluid property routines. When the developer is certain that the fluid will remain a liquid and wishes to abide by that assumption (generally, this is the case for most larger loops), all code from such modules should access the Glycol class of fluid property routines. Each of these classes will be described in separate sections below since each have different subroutine access to the main module.

Internally, both the refrigerant and glycol classes of fluids use “table lookup” and interpolation to find the appropriate value of a fluid property. No curve fits are done internally and the interpolation search routines are currently not optimized (no interval halving or special fast searching techniques are used to find the values).

Using Fluid Property Routines in EnergyPlus Modules

The routines are contained within a single module: **FluidProperties.f90**

Developers can use the routines anywhere inside EnergyPlus through the following USE statement:

USE FluidProperties

Access to this module may be limited by expanding this line of code with the ONLY designator.

Fluid Properties Functions for Refrigerant Class Fluids

In EnergyPlus, a refrigerant fluid is capable of being either in the liquid or vapor phase. Due to this definition, data must be available for both of these regions in order for the program to accurately calculate the various fluid properties. There are eight possible

functions that may be used to obtain refrigerant data using the Fluid Properties module. They include:

```
REAL FUNCTION GetSatPressureRefrig(Refrigerant,Temperature,RefrigIndex)
REAL FUNCTION GetSatTemperatureRefrig (Refrigerant,Pressure,RefrigIndex)
REAL FUNCTION GetSatEnthalpyRefrig (Refrigerant,Temperature,Quality,RefrigIndex)
REAL FUNCTION GetSatDensityRefrig (Refrigerant,Temperature,Quality,RefrigIndex)
REAL FUNCTION GetSatSpecificHeatRefrig (Refrigerant,Temperature,Quality,RefrigIndex)
REAL FUNCTION GetSupHeatEnthalpyRefrig (Refrigerant,Temperature,Pressure,RefrigIndex)
REAL FUNCTION GetSupHeatPressureRefrig (Refrigerant,Temperature,Enthalpy,RefrigIndex)
REAL FUNCTION GetSupHeatDensityRefrig (Refrigerant,Temperature,Pressure,RefrigIndex)
```

While most of the variables passed into the routine are self-explanatory, the two variables that are common to each of these functions are Refrigerant and RefrigIndex. Refrigerant in this case is the character string name of the refrigerant in question as listed in the input file using the FluidNames object. This must be passed into the function to identify the fluid being referenced. RefrigIndex is an internal variable. On the first call to the fluid property routine, it is zero. All of the fluid property routines are set-up to find a non-zero index in the local fluid property data structure that corresponds to this refrigerant name. Since finding the proper fluid from the fluid name each and every time is computationally slow, the index allows the code to quickly find the right data without doing an inordinate number of string comparisons. Thus, module developers should store the RefrigIndex in their own local data structure in addition to the refrigerant name.

Units for these other variables in these function calls are: Joules per kilogram for enthalpy, degrees Celsius for temperature, Pascals for pressure, kilograms per cubic meter for density, and Joules per kilogram-degree Celsius for specific heat. Quality and concentration are dimensionless fractions. All variables are considered input variables.

Module developers should use the functions listed above to first determine whether they are in the saturated region or the superheated region. The GetSatPressureRefrig and GetSatTemperatureRefrig functions should assist the users in determining whether they are in or beyond the saturated region. Once this is determined, the developer can call the appropriate function to obtain the quantity of interest: in the saturated region this includes the enthalpy, density, or specific heat; in the superheated region this includes the enthalpy, pressure, or density.

Reference Data Set (RDS) Values for Refrigerant Class Fluids

The data for refrigerants that are included in the reference data set that comes with EnergyPlus are as follows (temperatures in Celsius, pressure in MegaPascals):

Table 2. Regions for Fluid Properties

<u>Refrigerant</u>	<u>Sat. Temp range</u>	<u>Super Temp range*</u>	<u>Super Pressure range*</u>
R11	-70 to 198	-56.67 to 287.78	.001 to 1.379
R11(specheat)	-73 to 187		
R12	-100 to 111.8	-63 to 297	.01 to 8.0

R12(specheat)	-103 to 97		
R22	-90 to 96.15	-67.78 to 243.33	.021 to 3.31
R22(specheat)	-103 to 87		
NH3	-77.67 to 132.3	-45.6 to 198.9	.0345 to 1.93
NH3(specheat)	-73 to 127		
Steam	-60 to 200	100 to 1300	.01 to 60.0
Steam(specheat)	0 to 100		

*Obviously data for all temperatures at all pressures isn't loaded. The entire range of pressures given above will work, but the temperature range for a given pressure will be some subset of the Super Temp range shown above.

Subcooled region actually only returns h(f) or the saturated liquid value at the temperature you input.

Fluid Property Data and Expanding the Refrigerants Available to EnergyPlus

The Fluid Property routines have been reengineered to allow other users to add refrigerants to the input file without having to make any changes to the program code. The only requirement on input is that in order to add a new refrigerant, a user must enter a full set of data. The exact definition of a full set of data is given below.

As with all EnergyPlus input, the fluid properties data has both an input data description and a reference data set that must show up in the input file. All of the "standard" refrigerants list above must show up in the in.idf file for it to be available to the rest of the simulation. Below is the description of the input data description syntax for the fluid properties entries.

The first syntax item lists all of the fluids present in an input file and categorizes them as either a refrigerant (such as R11, R12, etc.) or a glycol (such as ethylene glycol, propylene glycol, etc.). A refrigerant or glycol must show up in this list in order to used as a valid fluid in other loops in the input file.

```
FluidNames,
    \unique-object
    \memo list of potential fluid names/types in the input file, maximum of ten
A1, \field fluid name 1
    \type alpha
A2, \field type of fluid for fluid name 1
    \type choice
    \key REFRIGERANT
    \key GLYCOL
A3, \field fluid name 2
    \type alpha
A4, \field type of fluid for fluid name 2
    \type choice
    \key REFRIGERANT
    \key GLYCOL
... same thing repeated over and over again ...
A19, \field fluid name 10
    \type alpha
A20; \field type of fluid for fluid name 10
    \type choice
    \key REFRIGERANT
    \key GLYCOL
```

An example of this statement in an input data file is:

```
FluidNames,
  R11, REFRIGERANT,
  R12, REFRIGERANT,
  R22, REFRIGERANT,
  NH3, REFRIGERANT,
  Steam, REFRIGERANT,
  NewGlycol, GLYCOL,
  SuperGlycol, GLYCOL;
```

All fluid properties vary with temperature. As a result, the following syntax allows the user to list the temperatures at which the data points are valid. Since in many cases, the temperatures will be similar, this provides a more compact input structure and avoids listing the temperatures multiple times. The name associated with the temperature list is the piece of information that will allow the actual fluid property data statements to refer back to or link to the temperatures. Up to 250 points may be entered with this syntax and temperatures must be entered in ascending order. Units for the temperatures are degrees Celsius. The same temperature list may be used by more than one refrigerant.

```
FluidPropertyTemperatures,
  \memo property values for fluid properties
  \memo list of up to 250 temperatures, note that number of property values must match the
number of properties
  \memo in other words, there MUST be a one-to-one correspondance between the property values
in this list and
  \memo the actual properties list in other syntax
  \units degrees C (for all temperature inputs)
  A1, \field temperature list name
  \type alpha
  N1, \field temperature 1
  \type real
  N2, \field temperature 2
  \type real
  . . . same thing repeated over and over again . . .
  N250; \field temperature 250
  \type real
```

An example of this statement in an input data file is:

```
FluidPropertyTemperatures,
  R11Temperatures,
  -70,-65,-60,-55,-50,-45,-40,-35,-30,-25,-20,-15,-10,-5,0,2,4,6,8,10,12,14,16,18,
  20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,55,60,65,70,75,80,85,90,95,100,
  105,110,115,120,125,130,135,140,145,150,155,160,165,170,175,180,185,190,198;
```

Property data for the saturated region is entered with the following syntax. Before the actual data is entered, this line of input must identify the refrigerant the data is to be associated with, what the data represents (choice of one of three keywords), the phase of the data (either fluid or the difference between fluid and gas), and the temperature list reference that links each data point with a temperature.

```

FluidPropertySaturated,
  \memo fluid properties for the saturated region
A1, \field fluid name (R11, R12, etc.)
  \reference FluidNames
A2, \field fluid property type
  \type choice
  \key ENTHALPY      ! Units are J/kg
  \key DENSITY       ! Units are J/kg-K
  \key SPECIFICHEAT ! Units are kg/m^3
A3, \field fluid phase
  \type choice
  \key FLUID         ! saturated fluid
  \key FLUIDGAS      ! difference between saturated fluid and saturated vapor
A4, \field temperatures list name
  \reference FluidPropertyTemperatures
N1, \field property value 1
  \type real
N2, \field property value 2
  \type real
. . . same thing repeated over and over again . . .
N250; \field property value 250
      \type real

```

An example of this statement in an input data file is:

```

FluidPropertySaturated,
  R11,ENTHALPY,FLUID,R11Temperatures, ! Enthalpy in J/kg
  153580,154600,156310,158580,161300,164380,167740,171330,175100,179020,183060,
  187190,191400,195680,200000,201740,203490,205240,207000,208770,210530,212310,
  214080,215870,217650,219860,221230,223030,224830,226630,228860,230250,232060,
  233860,235700,237520,239350,241180,243010,246350,249450,254080,258730,263480,
  268110,272860,277000,282410,287240,292120,297030,302000,307090,312080,317210,
  322400,327670,333020,338460,344010,349680,355500,361480,367690,374100,381060,
  388850,397280,426300;

```

The format of the data for the superheated region is almost identical to that of the saturated region with one addition—a pressure. The pressure is listed before the rest of the data and has units of Pa.

```

FluidPropertySuperheated,
  \memo fluid properties for the saturated region
A1, \field fluid name (R11, R12, etc.)
  \reference FluidNames
A2, \field fluid property type
  \type choice
  \key ENTHALPY      ! Units are J/kg
  \key DENSITY       ! Units are J/kg-K
  \key SPECIFICHEAT ! Units are kg/m^3 (Currently this data is not used, entered,
                                     or expected)
A3, \field temperatures list name
  \reference FluidPropertyTemperatures
N1, \field pressure
  \memo pressure for this list of properties
  \type real
  \units Pa
  \minimum> 0.0
N2, \field property value 1
  \type real
N3, \field property value 2
  \type real
. . . same thing repeated over and over again . . .
N251; \field property value 250
      \type real

```

An example of this statement in an input data file is:

```
FluidPropertySuperheated,
  R11,DENSITY,SuperR11Temperatures, ! Density in kg/m^3
  62000., !Pressure = 62000Pa
  0,0,0,0,0,0,0,0.0139,0.0134,0.0129,0.0124,0.012,0.0116,0.0112,0.0109,0.0105,
  0.0102,0.0099,0.0097,0.0094,0.0092,0.0089,0,0,0,0,0,0,0,0,0,0;
```

Fluid Properties Functions for Glycol Class Fluids

In EnergyPlus, a glycol fluid is assumed to remain in the liquid phase. As a result, data is only required for fluids in the liquid state. There are four possible functions that may be used to obtain glycol data using the Fluid Properties module. These correspond to the fluid property of interest and include:

REAL FUNCTION GetSpecificHeatGlycol (Glycol,Temperature,GlycolIndex)

REAL FUNCTION GetConductivityGlycol (Glycol,Temperature,GlycolIndex)

REAL FUNCTION GetDensityGlycol (Glycol,Temperature,GlycolIndex)

REAL FUNCTION GetViscosityGlycol (Glycol,Temperature,GlycolIndex)

All of these functions are used in exactly the same way. The module developer should send the glycol name (as listed in the GlycolConcentrations object in the input file) to the routine and the GlycolIndex (sent as 0 the first time and then set by the fluid property routine; see RefrigIndex discussion above). In addition, the functions require the temperature of the glycol in degrees Celsius.

Default Values for Glycol Class Fluids

With Version 1.2.1, EnergyPlus implemented default values for specific heat, density, conductivity, and viscosity for Water, Ethylene Glycol, and Propylene Glycol. This means that if users accept the values as published in the ASHRAE Handbook of Fundamentals, then the only information the user must include in their input file is a description of the concentration of glycol used (via the GlycolConcentrations object). If water is used in a loop, the user does not need to enter anything other than WATER as the fluid type in the appropriate input syntax. Data for various concentrations of these three default fluids encompasses the range over with these fluids and their combinations are in the liquid phase (-35 to 125 degrees Celsius). When the glycol combination in question is indeed a fluid, the functions will return the appropriate value for the parameter in question. If the glycol is either a solid or vapor, the routine will return a zero value. Units for the different properties are: Joules per kilogram-Kelvin for specific heat, Pascal-seconds for viscosity, Watts per meter-Kelvin for conductivity, and kilograms per cubic meter for density. In contrast to the refrigerant data which is included in the RDS and must be copied into the user input file if it is to be used, the glycol default data has been hardwired into EnergyPlus and does not need to be entered into the input file.

Fluid Property Data and Expanding the Glycols Available to EnergyPlus

The format of the data for the glycols is almost identical to that of the superheated region for refrigerants with one exception—concentration replaces pressure. The concentration is listed before the rest of the data and is dimensionless.


```

FluidPropertyConcentration,
  \memo fluid properties for water/other fluid mixtures
A1, \field fluid name (ethylene glycol, etc.)
  \reference FluidNames
A2, \field fluid property type
  \type choice
  \key DENSITY      ! Units are kg/m3
  \key SPECIFICHEAT ! Units are J/kg-K
  \key CONDUCTIVITY ! Units are W/m-K
  \key VISCOSITY    ! Units are N-s/m2
A3, \field temperatures list name
  \reference FluidPropertyTemperatures
N1, \field concentration
  \memo glycol concentration for this list of properties
  \type real
  \units percentage (as a real decimal)
  \minimum 0.0
  \maximum 1.0
N2, \field property value 1
  \type real
N3, \field property value 2
  \type real
. . . same thing repeated over and over again . . .
N251; \field property value 250
  \type real

```

An example of this statement in an input data file is:

```

FluidPropertyConcentration,
  MyPropyleneGlycol,SPECIFICHEAT ,GlycolTemperatures, ! Specific heat in J/kg-K
  0.8, ! Concentration
  2572,2600,2627,2655,2683,2710,2738,2766,2793,2821,2849,2876,2904,2931,2959,
  2987,3014,3042,3070,3097,3125,3153,3180,3208,3236,3263,3291,3319,3346,3374,
  3402,3429,3457;

```

The above input syntax is used to define data for a particular new fluid beyond the default glycol fluids. It would be repeated at other appropriate concentration values, if necessary, to define the fluid. It should be noted that in order to enter a fluid, the user must specify all four of the properties: conductivity, specific heat, viscosity, and density.

In addition to specifying the raw data for a new glycol, the user must list the fluid in the FluidNames object and then specify the concentration in the GlycolConcentrations object as shown below:

```

FluidNames,
  MyPropyleneGlycol, GLYCOL;
GlycolConcentrations,
  MyPropyleneGlycol, GLYCOL;

```

The IDD description for the GlycolConcentrations object is given below:

```

GlycolConcentrations,
    \unique-object
    \memo list of glycols and what concentration they are, maximum of ten
A1, \field fluid name 1
    \type alpha
A2, \field glycol name 1
    \type choice
    \key EthyleneGlycol
    \key PropyleneGlycol
    \memo or User Defined Fluid (must show up as a glycol in FluidNames list)
N1, \field concentration 1
    \type real
    \minimum 0.0
    \maximum 1.0
A3, \field fluid name 2
    \type alpha
A4, \field glycol name 2
    \type choice
    \key EthyleneGlycol
    \key PropyleneGlycol
    \memo or User Defined Fluid (must show up as a glycol in FluidNames list)
N2, \field concentration 2
    \type real
    \minimum 0.0
    \maximum 1.0
. . . repeated up to 10 times . . .
A19, \field fluid name 10
    \type alpha
A20, \field glycol name 10
    \type choice
    \key EthyleneGlycol
    \key PropyleneGlycol
    \memo or User Defined Fluid (must show up as a glycol in FluidNames list)
N10; \field concentration 10
    \type real
    \minimum 0.0
    \maximum 1.0

```

An example of how this would be used in an actual IDF is:

```

GlycolConcentrations,
MyProGly80Percent,  !- fluid name 1
MyPropyleneGlycol,  !- glycol name 1
0.8,
EthGly30Percent,    !- fluid name 2
EthyleneGlycol,     !- glycol name 2
0.3;  !- concentration 2

```

The key relationship in this syntax is how FluidNames relates to GlycolConcentrations and how to have modules access through the proper name. FluidNames are used to define raw data, whether for refrigerants or glycols. With a glycol, it is not enough to define raw data since this does not necessarily define the actual concentration of glycol being used. Thus, the GlycolConcentrations object is needed. It defines a name for the actual glycol and then refers back to the FluidNames (first fluid listed in the above example) or to one of the default glycol fluids (second fluid listed in the above example). It is critical that module developers refer to the "fluid name" listed in the GlycolConcentrations object. This is the name used inside the fluid property module to access the proper data. Note that when the GlycolConcentrations object is read in during execution that the module will interpolate down from a two-dimensional array of data (variation on temperature and concentration) to a one-dimensional array of data (with temperature as the only independent variable, concentration of a glycol fluid on any loop is assumed to be constant). This means that only the temperature (along with the glycol fluid name and index) must be passed into the fluid property module and also saves execution time since only a one-dimensional interpolation is needed.

Weather Services

All weather data (including Design Day and Location validation) are processed by the WeatherManager module. The SimulationManager invokes the WeatherManager at the proper times to retrieve data. The WeatherManager will retrieve the proper data for the current timestep/hour/day/month from the proper data source (design day definition, weather data file). The WeatherManager puts weather-type data (outside dry bulb, outside wet bulb, humidity, barometric pressure) into the DataEnvironment global data area. There is no need for other modules to call the WeatherManager directly. However, if there is some weather-type data that is needed and not provided in the DataEnvironment global area, contact us.

Flags and Parameters

Parameters

Constants that might be useful throughout the program are defined as Fortran parameters in the DataGlobals data module. Examples include *PI*, *PiOvr2*, *DegToRadians*, and *MaxNameLength*. DataHVACGlobals contains parameters that might be useful anywhere in the HVAC simulation. Some examples are *SmallTempDiff* and *SmallMassFlow* that can be used for preventing divide by zero errors. The full set of global parameters can be obtained by examining the modules DataGlobals and DataHVACGlobals.

Simulation Flags

A number of logical flags (variables that are either *true* or *false*) are used throughout EnergyPlus. These flags are normally used to indicate the start or end of a time or simulation period. The following shows a complete list.

In DataGlobals:

BeginSimFlag

Set to true until the actual simulation has begun, set to false after first heat balance time step.

BeginFullSimFlag

Set to true until a full simulation begins (as opposed to a sizing simulation); set to false after the first heat balance time step of the full simulation.

EndSimFlag

Normally false, but set to true at the end of the simulation (last heat balance time step of last hour of last day of last environment).

WarmupFlag

Set to true during the warmup portion of a simulation; otherwise false.

BeginEnvrnFlag

Set to true at the start of each environment (design day or run period), set to false after first heat balance time step in environment. This flag should be used for beginning of environment initializations in most HVAC components. See the example module for correct usage.

EndEnvrnFlag

Normally false, but set to true at the end of each environment (last heat balance time step of last hour of last day of environment).

BeginDayFlag

Set to true at the start of each day, set to false after first heat balance time step in day.

EndDayFlag

Normally false, but set to true at the end of each day (last heat balance time step of last hour of day).

BeginHourFlag

Set to true at the start of each hour, set to false after first heat balance time step in hour.

EndHourFlag

Normally false, but set to true at the end of each hour (last heat balance time step of hour)

BeginTimeStepFlag

Set to true at the start of each heat balance time step, set to false after first HVAC step in the heat balance time step.



e: never set to false![DBS2]

EndTimeStepFlag

Normally false, but set to true at the end of each heat balance time step.

Note: never set to true!

In DataHVACGlobals:**FirstTimeStepSysFlag**

Set to true at the start of the first HVAC time step within each heat balance time step, false at the end of the HVAC time step. In other words, this flag is true during the first HVAC time step in a heat balance time step, and is false otherwise.

BeginAirLoopFlag

True for first SimAirLoop call; false thereafter.

Note: not used and not set!

In Subroutine SimHVAC:**FirstHVACIteration**

True when HVAC solution technique on first iteration, false otherwise.
Passed as a subroutine argument into the HVAC equipment simulation driver routines.

The most commonly used logical flag in the HVAC simulation is FirstHVACIteration that is passed around as an argument among the HVAC simulation subroutines. The HVAC simulation is solved iteratively each HVAC time step. FirstHVACIteration is true for the first iteration in each time step and false for the remaining iterations.

Finally, each developer must define and set a "GetInput" flag to make sure input data is read in only once. In the example module Fans the GetInput flag is GetInputFlag; the new developer can follow this example in using such a flag.

Psychrometric services

EnergyPlus has a full complement of psychrometric functions. All the routines are Fortran functions returning a single precision real value. All arguments and results are in SI units.

The Names for the different Psychrometric Routines are based on the following self-explanatory format; the different variables used in the Psych Routine taxonomy are as follows.

- H = Enthalpy
- W= Humidity Ratio
- Rh= Relative Humidity
- V= Specific Volume
- Rhov= Vapor Density of Air
- Hfg = Latent energy (heat of vaporization for moist air)
- Hg= Enthalpy of gaseous moisture
- Pb= Barometric Pressure
- Twb=Temperature Wet Bulb
- Twd= Temperature Dry Bulb
- Tdp= Temperature Dew Point
- Tsat and Psat= Saturation Temperature and Saturation Pressure
- Psy## Fn ## = Psy {## is a Function of ##}
- Note: Each of the two capital alphabets together have different meaning
Eg: **{Psy ## Fn HW}= {Psy ## Function of *Enthalpy and Humidity Ratio*}**

PsyRhoAirFnPbTdbW (Pb,Tdb,W)

Returns the density of air in kilograms per cubic meter as a function of barometric pressure [Pb] (in Pascals), dry bulb temperature [Tdb] (in Celsius), and humidity ratio [W] (kilograms of water per kilogram of dry air).

PsyCpAirFnWTdb (W,Tdb)

Returns the specific heat of air in Joules per kilogram degree Celsius as a function of humidity ratio [W] (kilograms of water per kilogram of dry air) and dry bulb temperature [Tdb] (Celsius).

PsyHfgAirFnWTdb (W,Tdb)

Returns the Latent energy of air [Hfg](Joules per kilogram) as a function of humidity ratio [W] (kilograms of water per kilogram of dry air) and dry bulb temperature [Tdb] (Celsius). It calculates hg and then hf and the difference is Hfg.

PsyHgAirFnWTdb (W,Tdb)

Returns the specific enthalpy of the moisture as a gas in the air in Joules per kilogram as a function of humidity ratio [W] (kilograms of water per kilogram of dry air) and dry bulb temperature [Tdb] (Celsius).

PsyTdpFnTdbTwbPb (Tdb,Twb,Pb)

Returns the dew point temperature in Celsius as a function of dry bulb temperature [Tdb] (Celsius), wet bulb temperature [Twb] (Celsius), and barometric pressure [Pb] (Pascals).

PsyTdpFnWPb (W,Pb)

Returns the dew point temperature in Celsius as a function of humidity ratio [W] (kilograms of water per kilogram of dry air) and barometric pressure [Pb] (Pascals).

PsyHFnTdbW (Tdb,W)

Returns the specific enthalpy of air in Joules per kilogram as a function of dry bulb temperature [Tdb] (Celsius) and humidity ratio [W] (kilograms of water per kilogram of dry air).

PsyHFnTdbRhPb (Tdb,Rh,Pb)

Returns the specific enthalpy of air in Joules per kilogram as a function of dry bulb temperature [Tdb] (Celsius), relative humidity [Rh] (fraction), and barometric pressure [Pb] (Pascals).

PsyTdbFnHW (H,W)

Returns the air temperature in Celsius as a function of air specific enthalpy [H] (Joules per kilogram) and humidity ratio [W] (kilograms of water per kilogram of dry air).

PsyRhovFnTdbRh (Tdb,Rh)

Returns the Vapor Density in air [RhoVapor](kilograms of water per cubic meter of air) as a function of dry bulb temperature [Tdb](Celsius), Relative Humidity [Rh] (fraction).

PsyRhovFnTdbWP (Tdb,W,Pb)

Returns the Vapor Density in air [RhoVapor](kilograms of water per cubic meter of air) as a function of dry bulb temperature [Tdb](Celsius), humidity ratio [W] (kilograms of water per kilogram of dry air) and barometric pressure [Pb] (Pascals).

PsyRhFnTdbRhov (Tdb,Rhov)

Returns the Relative Humidity [Rh] (fraction) in air as a function of dry bulb temperature [Tdb] (Celsius) and Vapor Density in air [RhoVapor](kilograms of water per cubic meter of air).

PsyRhFnTdbWPb (Tdb,W,Pb)

Returns the relative humidity (fraction) as a function of dry bulb temperature [Tdb] (Celsius), humidity ratio [W] (kilograms of water per kilogram of dry air) and barometric pressure [Pb] (Pascals).

PsyTwbFnTdbWPb (Tdb,W,Pb)

Returns the air wet bulb temperature in Celsius as a function of dry bulb temperature [Tdb] (Celsius), humidity ratio [W] (kilograms of water per kilogram of dry air) and barometric pressure [Pb] (Pascals).

PsyVFnTdbWPb (Tdb,W,Pb)

Returns the specific volume in cubic meters per kilogram as a function of dry bulb temperature [Tdb] (Celsius), humidity ratio [W] (kilograms of water per kilogram of dry air) and barometric pressure [Pb] (Pascals).

PsyWFnTdpPb (Tdp,Pb)

Returns the humidity ratio in kilograms of water per kilogram of dry air as a function of the dew point temperature [Tdp] (Celsius) and barometric pressure [Pb] (Pascals).

PsyWFnTdbH (Tdb,H)

Returns the humidity ratio in kilograms of water per kilogram of dry air as a function of dry bulb temperature [Tdb] (Celsius) and air specific enthalpy [H] (Joules per kilogram).

PsyWFnTdbTwbPb (Tdb,Twb,Pb)

Returns the humidity ratio in kilograms of water per kilogram of dry air as a function of dry bulb temperature [Tdb] (Celsius), wet bulb temperature [Twb] (Celsius), and barometric pressure [Pb] (Pascals).

PsyWFnTdbRhPb (Tdb,Rh,Pb)

Returns the humidity ratio in kilograms of water per kilogram of dry air as a function of dry bulb temperature [Tdb] (Celsius), relative humidity [RH] (fraction), and barometric pressure [Pb] (Pascals).

PsyPsatFnTemp (T)

Returns the saturation pressure in Pascals as a function of the air saturation temperature [T] (Celsius).

PsyTsatFnHPb (H,Pb)

Returns the air saturation temperature in Celsius as a function of air specific enthalpy [H] (Joules per kilogram) and barometric pressure [Pb] (Pascals).

PsyTsatFnPb (P)

Returns the air saturation temperature in Celsius as a function of saturation pressure [P] (Pascals).

CPCW (Temp)

Returns Specific heat capacity (Joule/kilogram/kelvin) for chilled water as function of temperature [T] (Celsius).

CPHW (Temp)

Returns Specific heat capacity (Joule/kilogram/kelvin) for hot water as function of temperature [T] (Celsius).

CVHW (Temp)

Returns Specific heat capacity (Joule/kilogram/kelvin) for hot water at constant volume as function of temperature [T] (Celsius).

RhoH2O (Temp)

Returns density of water (kg/m3) as function of Temperature [T] (Celsius).

Tabular Output Utilities

Several utility routines are available to help generate tabular reports. To create tabular reports, the developer needs to create a routine called something like WriteTabularX. The WriteTabularX routine should appear in SimulationManger between the OpenOutputTabularFile and CloseOutputTabularFile calls. The WriteTabularX routine should make use of several utilities described below. The "USE" statement reference OutputReportTabular module. Good example of how to use this facility are in the OutputReportTabular file and the EconomicTariff file.

WriteReportHeaders(reportName,objectName,averageOrSum)

Where reportName is the name that you want the report to be called and the objectName is the name of the object that appears after the "For: " for each instance of the report. The averageOrSum flag when set to SUM adds the phrase "per second" after the reportName.

WriteSubtitle(subtitle)

Where the subtitle is a string that usually appears before a specific table. This is useful if the report includes multiple tables.

WriteTable(body,rowLabels,columnLabels,widthColumn)

The WriteTable routine actually generates the tables that appear in the tabular output file (CSV, HTML, or TXT). The rowLabels and columnLables are both one dimensional string arrays that contain the appropriate labels. If the column labels strings include the vertical bar symbol "|" then when creating a text report, the labels will be split between lines at the vertical bar. For HTML and CSV output, the vertical bar symbols are removed prior to display.

The body array is a two dimensional array (row,column order) containing the cells in the body of the table. It must be strings so conversion utilities such as RealToStr should be used to convert from numeric values.

WidthColumn is a one dimensional integer array containing the column widths for use only with the fixed width text output option.

HVAC Network

Branches, Connectors, and Nodes

In EnergyPlus, the HVAC system and plant form a network (technically, a graph). The individual pieces of equipment – the fans, coils, chillers, etc. – are connected together by air ducts and fluid pipes. In EnergyPlus nomenclature, the air and fluid circuits are called loops. Specifying how an individual system and plant are connected is done in the EnergyPlus input (IDF) file. The overall structure of the network is defined with Branch and Connector objects. The detail is filled with components and their inlet and outlet nodes. A Branch consists of one or more components arranged sequentially along a pipe or duct. A Connector specifies how three or more branches are connected through a Splitter or Mixer. Nodes connect components along a branch: the outlet node of one component is the inlet node of the next downstream component. The nodes represent conditions at a point on a loop. Each component has one or more inlet and outlet nodes, depending on how many loops it interacts with. A fan, for instance, has one inlet node and one outlet node, since it interacts with a single air loop. A water coil will have 2 inlet and 2 outlet nodes, since it interacts with an air and a fluid loop. Figure 1 shows a diagram of an EnergyPlus HVAC input.

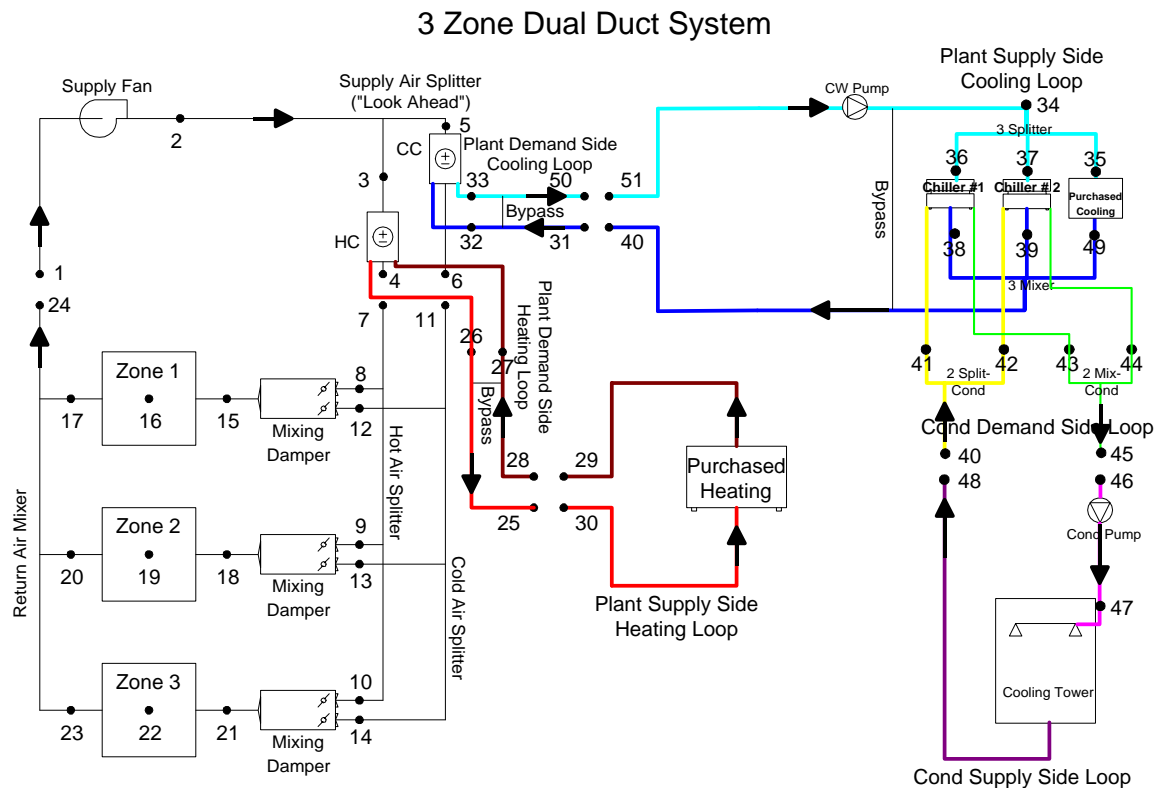


Figure 1. HVAC Input Diagram

As an illustration of how such a network is built up on the IDF, here is the section of the IDF that describes the supply fan, splitter, and heating and cooling coil section of the dual duct air system.

```

BRANCH LIST,
  Dual Duct Air Loop Branches , ! Branch List Name
  Air Loop Main Branch ,      ! Branch Name 1
  Heating Coil Air Sys Branch ,      ! Branch Name 2
  Cooling Coil Air Sys Branch ;      ! Branch Name 3

CONNECTOR LIST,
  Dual Duct Connectors , ! Connector List Name
  SPLITTER ,            ! Type of Connector 1
  DualDuctAirSplitter ; ! Name of Connector 1

BRANCH,
  Air Loop Main Branch , ! Branch Name
  1.3 ,                  ! Maximum Branch Flow Rate
  FAN:SIMPLE:ConstVolume, Supply Fan 1,
    Supply Fan Inlet Node, Supply Fan Outlet Node, PASSIVE ;

BRANCH,
  Heating Coil Air Sys Branch , ! Branch Name
  1.3 ,                          ! Maximum Branch Flow Rate
  COIL:Water:SimpleHeating, Main Heating Coil,
    Heating Coil Inlet Node, Heating Coil Outlet Node, ACTIVE ;

BRANCH,
  Cooling Coil Air Sys Branch , ! Branch Name
  1.3 ,                          ! Maximum Branch Flow Rate
  COIL:Water:DetailedFlatCooling, Detailed Cooling Coil,
    Cooling Coil Inlet Node, Cooling Coil Outlet Node, ACTIVE ;

SPLITTER,
  DualDuctAirSplitter,
  Air Loop Main Branch,
  Heating Coil Air Sys Branch,
  Cooling Coil Air Sys Branch;

FAN:SIMPLE:ConstVolume,
  Supply Fan 1, !Fan Name
  FanAndCoilAvailSched, !Fan Schedule
  0.7,           !Fan Efficiency
  600.0,         !Delta Pressure [N/M^2]
  1.3,           !Max Vol Flow Rate [m^3/Sec]
  0.9,           !motor efficiency
  1.0,           !motor in air stream fraction
  Supply Fan Inlet Node, ! Inlet Node
  Supply Fan Outlet Node; ! Outlet Node

COIL:Water:DetailedFlatCooling,
  Detailed Cooling Coil, !Name of cooling coil
  CoolingCoilAvailSched, !Cooling Coil Schedule
  1.1,                   !Max Water Flow Rate of Coil kg/sec
  6.23816,               !Tube Outside Surf Area
  6.20007018,           !Tube Inside Surf Area
  101.7158224,          !Fin Surf Area
  0.300606367,          !Min Air Flow Area
  0.165097968,          !Coil Depth
  0.43507152,           !Coil Height
  0.001499982,          !Fin Thickness
  0.014449958,          !Tube Inside Diameter
  0.015879775,          !Tube Outside Diameter
  0.385764854,          !Tube Thermal Conductivity
  0.203882537,          !Fin Thermal Conductivity

```

```

0.001814292,      !Fin Spacing
0.02589977,       !Tube Depth
6,                !Number of Tube Rows
16,               !Number of Tubes per Row
Cooling Coil Water Inlet Node, ! Coil Water Side Inlet
Cooling Coil Water Outlet Node, ! Coil Water Side Outlet
Cooling Coil Inlet Node, ! Coil Air Side Inlet
Cooling Coil Outlet Node; ! Coil Air Side Outlet

COIL:Water:SimpleHeating,
    Main Heating Coil,      !Name of heating coil
    FanAndCoilAvailsSched, !heating Coil Schedule
    1200.0,                 !UA of the Coil
    4.3,                   !Max Water Flow Rate of Coil kg/sec
    Heating Coil Water Inlet, ! Coil Water Side Inlet
    Heating Coil Water Outlet, ! Coil Water Side Outlet
    Heating Coil Inlet Node, ! Coil Air Side Inlet
    Heating Coil Outlet Node; ! Coil Air Side Outlet

```

Obviously, the creation of such a system/plant network description is best handled by a graphical user interface (GUI). However, for testing purposes a developer may have to create the input for a component by hand and insert it into an existing IDF. Then the developer must be careful to choose unique names for the branches and nodes and make sure the entire network makes physical sense.

Nodes in the simulation

In the EnergyPlus data structure, the nodes are where each component model gets its input and where it places its output. The module *DataLoopNode* contains all the node related data. In particular, the array *Node* contains the state variables and mass flows for all the nodes in the problem being simulated.

```

TYPE NodeData
    CHARACTER(len=MaxNameLength) :: FluidType      ! must be one of the
valid parameters, AirNode, WaterNode
    REAL :: Temp = 0.0 !
    REAL :: TempMin = 0.0 !
    REAL :: TempMax = 0.0 !
    REAL :: TempSetPoint = 0.0 !
    REAL :: MassFlowRate = 0.0 !
    REAL :: MassFlowRateMin = 0.0 !
    REAL :: MassFlowRateMax = 0.0 !
    REAL :: MassFlowRateMinAvail = 0.0 !
    REAL :: MassFlowRateMaxAvail = 0.0 !
    REAL :: MassFlowRateSetPoint = 0.0 !
    REAL :: Quality = 0.0 !
    REAL :: Press = 0.0 !
    REAL :: Enthalpy = 0.0 !
    REAL :: HumRat = 0.0 !
    REAL :: HumRatMin = 0.0 !
    REAL :: HumRatMax = 0.0 !
    REAL :: HumRatSetPoint = 0.0 !
END TYPE NodeData

TYPE MoreNodeData
    REAL :: ReportEnthalpy = 0.0 ! specific enthalpy
calculated at the HVAC timestep [J/kg]
    REAL :: VolFlowRate = 0.0 ! volume flow rate
[m3/s]
    REAL :: WetbulbTemp = 0.0 ! wetbulb temperature
[C]

```

```

END TYPE MoreNodeData
TYPE (NodeData), ALLOCATABLE, DIMENSION(:) :: Node !dim to num nodes
in SimHVAC
TYPE (MoreNodeData), ALLOCATABLE, DIMENSION(:) :: MoreNodeInfo

```

In our example module *NewHVACComponent*, the subroutine *InitNewHVACComponent* is responsible for obtaining the input data from the inlet node(s) and putting it into the component data structure for use in *CalcNewHVACComponent*. Then *UpdateNewHVACComponent* takes the calculated data and moves it to the outlet nodes for use by other components. EnergyPlus component models are assumed to be direct models: inlets are input to the calculation and outlets are output from the calculations.

Getting Nodes

Data Flow in an HVAC Component Module

The data in an EnergyPlus HVAC component module resides in three places.

1. The component inlet nodes – this is where the data input to the model resides.
2. The component internal data structure(s) – one or more arrays of data structures which contain all the data needed for the component simulation. This includes data from the input file, data from the inlet nodes, and any schedule values. In addition, these data structure(s) store the results of the calculation.
3. The component outlet nodes – data is moved from the internal data structure(s) to the outlet nodes at the completion of each component simulation.

The data flows from the inlet nodes into the component internal data structure(s) and then into the outlet nodes. Let us see how this works in our example module Fans.

At the start of the module, the component internal data structure is defined.

```

TYPE FanEquipConditions
  CHARACTER(len=MaxNameLength) :: FanName ! Name of the fan
  CHARACTER(len=MaxNameLength) :: FanType ! Type of Fan ie. Simple, Vane axial, Centrifugal, etc.
  CHARACTER(len=MaxNameLength) :: Schedule ! Fan Operation Schedule
  CHARACTER(len=MaxNameLength) :: Control ! ie. Const Vol, Variable Vol
  Integer :: SchedPtr ! Pointer to the correct schedule
  REAL :: InletAirMassFlowRate !MassFlow through the Fan being Simulated [kg/Sec]
  REAL :: OutletAirMassFlowRate
  Real :: MaxAirFlowRate !Max Specified Volume Flow Rate of Fan [m^3/sec]
  Real :: MinAirFlowRate !Min Specified Volume Flow Rate of Fan [m^3/sec]
  REAL :: MaxAirMassFlowRate ! Max flow rate of fan in kg/sec
  REAL :: MinAirMassFlowRate ! Min flow rate of fan in kg/sec
  REAL :: InletAirTemp
  REAL :: OutletAirTemp
  REAL :: InletAirHumRat
  REAL :: OutletAirHumRat
  REAL :: InletAirEnthalpy
  REAL :: OutletAirEnthalpy
  REAL :: FanPower !Power of the Fan being Simulated [kW]
  REAL :: FanEnergy !Fan energy in [kJ]
  REAL :: DeltaTemp !Temp Rise across the Fan [C]
  REAL :: DeltaPress !Delta Pressure Across the Fan [N/M^2]
  REAL :: FanEff !Fan total efficiency; motor and mechanical
  REAL :: MotEff !Fan motor efficiency
  REAL :: MotInAirFrac !Fraction of motor heat entering air stream
  REAL, Dimension(5):: FanCoeff !Fan Part Load Coefficients to match fan type
  ! Mass Flow Rate Control Variables
  REAL :: MassFlowRateMaxAvail
  REAL :: MassFlowRateMinAvail
  INTEGER :: InletNodeNum
  INTEGER :: OutletNodeNum
END TYPE FanEquipConditions

!MODULE VARIABLE DECLARATIONS:
  INTEGER :: NumFans ! The Number of Fans found in the Input
  TYPE (FanEquipConditions), ALLOCATABLE, DIMENSION(:) :: Fan

```

In this case, there is only one structure that stores all of the fan data. We could have chosen to divide this rather large structure up into separate structures – one for input file data, one for inlet data, and one for outlet data, for instance. Note that in Fortran 90 structures are called defined type. The TYPE – END TYPE construct defines a new data structure. Then an allocatable array *Fan* of the defined type is created. This one-dimensional array will contain an entry for each fan in the problem.

The internal data array is allocated (sized) in the “GetInput” routine GetFanInput.

```

NumSimpFan = GetNumObjectsFound('FAN:SIMPLE:CONSTVOLUME')
NumVarVolFan = GetNumObjectsFound('FAN:SIMPLE:VARIABLEVOLUME')
NumOnOff = GetNumObjectsFound('FAN:SIMPLE:ONOFF')
NumZoneExhFan = GetNumObjectsFound('ZONE EXHAUST FAN')
NumFans = NumSimpFan + NumVarVolFan + NumZoneExhFan+NumOnOff
IF (NumFans.GT.0) ALLOCATE(Fan(NumFans))

```

The remainder of the “GetInput” routine moves input file data into the Fan array. The “Init” routine transfers data from the inlet nodes into the same array in preparation for performing the calculation.

```

! Load the node data in this section for the component simulation
!
!First need to make sure that the massflowrate is between the max and min avail.
IF (Fan(FanNum)%FanType .NE. 'ZONE EXHAUST FAN') THEN
    Fan(FanNum)%InletAirMassFlowRate = Min(Node(InletNode)%MassFlowRate, &
                                           Fan(FanNum)%MassFlowRateMaxAvail)
    Fan(FanNum)%InletAirMassFlowRate = Max(Fan(FanNum)%InletAirMassFlowRate, &
                                           Fan(FanNum)%MassFlowRateMinAvail)
ELSE ! zone exhaust fans - always run at the max
    Fan(FanNum)%MassFlowRateMaxAvail = Fan(FanNum)%MaxAirMassFlowRate
    Fan(FanNum)%MassFlowRateMinAvail = 0.0
    Fan(FanNum)%InletAirMassFlowRate = Fan(FanNum)%MassFlowRateMaxAvail
END IF

!Then set the other conditions
Fan(FanNum)%InletAirTemp      = Node(InletNode)%Temp
Fan(FanNum)%InletAirHumRat    = Node(InletNode)%HumRat
Fan(FanNum)%InletAirEnthalpy  = Node(InletNode)%Enthalpy

```

The “Calc” routines do the actual component simulation. All the data they need has been stored in the internal data array ready to be used. The results of the calculation are, in this case, stored in the same array. The “Calc” routine always does pure calculation/simulation – it never retrieves or stores data.

```

DeltaPress = Fan(FanNum)%DeltaPress
FanEff      = Fan(FanNum)%FanEff

! For a Constant Volume Simple Fan the Max Flow Rate is the Flow Rate for the fan
Tin         = Fan(FanNum)%InletAirTemp
Win         = Fan(FanNum)%InletAirHumRat
RhoAir      = Fan(FanNum)%RhoAirStdInit
MassFlow    = MIN(Fan(FanNum)%InletAirMassFlowRate, Fan(FanNum)%MaxAirMassFlowRate)
MassFlow    = MAX(MassFlow, Fan(FanNum)%MinAirMassFlowRate)
!
!Determine the Fan Schedule for the Time step
If( ( GetCurrentScheduleValue(Fan(FanNum)%SchedPtr)>0.0 .and. Massflow>0.0 .or. TurnFansOn .and.
Massflow>0.0) &
    .and. .NOT.TurnFansOff ) Then
    !Fan is operating
    Fan(FanNum)%FanPower = MassFlow*DeltaPress/(FanEff*RhoAir) ! total fan power
    FanShaftPower = Fan(FanNum)%MotEff * Fan(FanNum)%FanPower ! power delivered to shaft
    PowerLossToAir = FanShaftPower + (Fan(FanNum)%FanPower - FanShaftPower) * &
        Fan(FanNum)%MotInAirFrac
    Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy + PowerLossToAir/MassFlow
    ! This fan does not change the moisture or Mass Flow across the component
    Fan(FanNum)%OutletAirHumRat = Fan(FanNum)%InletAirHumRat
    Fan(FanNum)%OutletAirMassFlowRate = MassFlow
    Fan(FanNum)%OutletAirTemp = PsyTdbFnHW
    (Fan(FanNum)%OutletAirEnthalpy, Fan(FanNum)%OutletAirHumRat)
Else
    !Fan is off and not operating no power consumed and mass flow rate.
    Fan(FanNum)%FanPower = 0.0
    FanShaftPower = 0.0
    PowerLossToAir = 0.0
    Fan(FanNum)%OutletAirMassFlowRate = 0.0
    Fan(FanNum)%OutletAirHumRat = Fan(FanNum)%InletAirHumRat
    Fan(FanNum)%OutletAirEnthalpy = Fan(FanNum)%InletAirEnthalpy
    Fan(FanNum)%OutletAirTemp = Fan(FanNum)%InletAirTemp
    ! Set the Control Flow variables to 0.0 flow when OFF.
    Fan(FanNum)%MassFlowRateMaxAvail = 0.0
    Fan(FanNum)%MassFlowRateMinAvail = 0.0

End If

```

Finally, the “Update” routine (UpdateFan) moves the results from the internal data array into the outlet node(s).

```

OutletNode = Fan(FanNum)%OutletNodeNum
InletNode = Fan(FanNum)%InletNodeNum

! Set the outlet air nodes of the fan
Node(OutletNode)%MassFlowRate = Fan(FanNum)%OutletAirMassFlowRate
Node(OutletNode)%Temp = Fan(FanNum)%OutletAirTemp
Node(OutletNode)%HumRat = Fan(FanNum)%OutletAirHumRat
Node(OutletNode)%Enthalpy = Fan(FanNum)%OutletAirEnthalpy
! Set the outlet nodes for properties that just pass through & not used
Node(OutletNode)%Quality = Node(InletNode)%Quality
Node(OutletNode)%Press = Node(InletNode)%Press

! Set the Node Flow Control Variables from the Fan Control Variables
Node(OutletNode)%MassFlowRateMaxAvail = Fan(FanNum)%MassFlowRateMaxAvail
Node(OutletNode)%MassFlowRateMinAvail = Fan(FanNum)%MassFlowRateMinAvail

IF (Fan(FanNum)%FanType .EQ. 'ZONE EXHAUST FAN') THEN
    Node(InletNode)%MassFlowRate = Fan(FanNum)%InletAirMassFlowRate
END IF

```

For the last few lines of the Update routine above, a special case is made to place information back on the inlet node for Zone Exhaust Fan. You may want to clarify why this is being done, since the discussion above says that Update is supposed to move results from the internal data array to the outlet nodes. Without a discussion, this might be confusing to the reader...

Certain data items must always be transferred from inlet nodes to outlet nodes even if the data item is unaltered by the component model. The data items that must be transferred are:

1. Temp
2. HumRat
3. Enthalpy
4. Press
5. MassFlowRate
6. MassFlowRateMaxAvail
7. MassFlowRateMinAvail

Node Mass Flow Variables

The node mass flow variables merit a little more discussion. Five mass flow variables are defined at each node. They are: MassFlowRate, MassFlowRateMin, MassFlowRateMax, MassFlowRateMinAvail and MassFlowRateMaxAvail. These variables hold loop mass flow rate information according to the following definitions.

- MassFlowRate – this node variable holds the simulation mass flow rate for the current timestep. The component simulation retrieves this mass flow rate from its inlet node (“Init” routine) and uses it as the initial mass flow rate in the simulation. The component simulation may or may not change the mass flow rate. In any case, it writes the value out to the exit node in the “Update” routine.
- MassFlowRateMax, MassFlowRateMin – These node variables hold the maximum possible and the minimum allowable flow rates for a particular component. As such, they represent the “hardware limit” on the flow rate for the component. By

convention, these variables are stored at the component outlet node. Since components share their nodes (the outlet node of one component is the inlet node of the next component), the protocol must be strictly followed. These variables are set by each component at the beginning of the simulation and are never reset thereafter.

- **MassFlowRateMaxAvail, MassFlowRateMinAvail** – these node variables represent the *loop* maximum and minimum flow rate for the current configuration of the loop on which the component resides. All components should change the max/min available values so that they lie within the max/min flow rate range for the component. The component simulation reads the current loop min/max available flow rate from its inlet node (“Init” routine) and writes the updated min/max available flow rate to its outlet node (“Update” routine).

The component simulation retrieves **MassFlowRate**, **MassFlowRateMinAvail**, and **MassFlowRateMaxAvail** along with other node variables from its input node prior to the simulation of the component. The flow rate must be checked and if necessary adjusted prior to the simulation: the **MassFlowRate** must be bounded by **MassFlowRateMaxAvail** and **MassFlowRateMinAvail**, which in turn must be bounded by **MassFlowRateMax** and **MassFlowRateMin** for the component. The following steps should be followed in the initialization stage of the component simulation.

- a) Compare the **MassFlowRateMinAvail** and **MassFlowRateMaxAvail** retrieved from the input node with the **MassFlowRateMin** and **MassFlowRateMax** for your component. If either of the retrieved values is out of bounds, replace that value with either the Max or Min for the component.
- b) Compare the **MassFlowRate** retrieved from the inlet node with the **MassFlowRateMinAvail** and **MassFlowRateMaxAvail**. If **MassFlowRate** is not bounded by **MassFlowRateMinAvail** and **MassFlowRateMaxAvail**, reset **MassFlowRate** to the nearest boundary value.

If the component model calculates **MassFlowRate**, it must be bounded by **MassFlowRateMin** and **MassFlowRateMax**. Following the component simulation, the **MassFlowRate** and the bounding **MassFlowRateMinAvail** and **MassFlowRateMaxAvail** should be written to the outlet node along with the other updated loop variables.

Output

There are several output files available in EnergyPlus. As you can see in Appendix A, DataGlobals contains OutputFileStandard, OutputFileInits, and OutputFileDebug.

OutputFileDebug is initialized very early in the EnergyPlus execution and is available for any debugging output the developer might need.

OutputFileInits is intended for “one-time” outputs and has not been extensively used to date. The structure will be similar to the IDD/IDF structure in that there will be a “definition” line followed by the data being reported.

OutputFileStandard is the usual output file from EnergyPlus. You can read more details from the [Guide for Interface Developers](#) document. There is an Output section there as well as the keywords for obtaining those outputs.

How Do I Output My Variables?

Module developers are responsible for “setting” up the variables that will appear in the OutputFileStandard.

To do this is very simple. All you need to do is place a simple call to *SetupOutputVariable* into your module for each variable to be available for reporting. This call should be done only once for each Variable/KeyedValue pair (see below). For HVAC and Plant components, this call is usually at the end of the “GetInput” subroutine. See the example module for an illustration of this. Other calls in the simulation routines will invoke the EnergyPlus *OutputProcessor* automatically at the proper time to have the data appear in the OutputFileStandard.

For you the call is:

```
Call SetupOutputVariable(VariableName,ActualVariable, &
                        IndexTypeKey, VariableTypeKey,KeyedValue,ReportFreq &
                        ResourceTypeKey,EndUseKey,GroupKey)
```

Interface statements allow for the same call to be used for either real or integer “ActualVariable” variables. A few examples from EnergyPlus and then we will define the arguments:

```
CALL SetupOutputVariable('Outdoor Dry Bulb [C]', &
                        OutDryBulbTemp,'Zone', &
                        'Average','Environment')

CALL SetupOutputVariable('Mean Air Temperature[C]', &
                        ZnRpt(Loop)%MeanAirTemp,'Zone', &
                        'State',Zone(Loop)%Name)

CALL SetupOutputVariable('Fan Coil Heating Energy[J]', &
                        FanCoil(FanCoilNum)%HeatEnergy,'System', &
                        'Sum',FanCoil(FanCoilNum)%Name)

CALL SetupOutputVariable('Humidifier Electric Consumption[J]',
                        Humidifier(HumNum)%ElecUseEnergy, &
                        'System','Sum', &
                        Humidifier(HumNum)%Name,&
                        ResourceTypeKey='ELECTRICITY',&
                        EndUseKey = 'HUMIDIFIER',&
                        GroupKey = 'System')
```

SetupOutputVariable Arguments	Description
VariableName	String name of variable, units should be included in []. If no units, use []
ActualVariable	This should be the actual variable that will store the value. The OutputProcessor sets up a pointer to this variable, so it will need to be a SAVED variable if in a local routine. As noted in examples, can be a simple variable or part of an array/derived type.
IndexTypeKey	When this variable has its proper value. 'Zone' is used for variables that will have value on the global timestep (alias "HeatBalance"). 'HVAC' is used for variables that will have values calculated on the variable system timesteps (alias "System", "Plant")
VariableTypeKey	Two kinds of variables are produced. 'State' or 'Average' are values that are instantaneous at the timestep (zone air temperature, outdoor weather conditions). 'NonState' or 'Sum' are values which need to be summed for a period (energy).
KeyedValue	Every variable to be reported needs to have an associated keyed value. Zone Air Temperature is available for each Zone, thus the keyed value is the Zone Name.
ReportFreq	This optional argument should only be used during debugging of your module but it is provided for the developers so that these variables would always show up in the OutputFile. (All other variables must be requested by the user).
ResourceTypeKey	Meter Resource Type; an optional argument used for including the variable in a meter. The meter resource type can be 'Electricity', 'Gas', 'Coal', 'FuelOil#1', 'FuelOil#2', 'Propane', 'Water', or 'EnergyTransfer'.
EndUseKey	Meter End Use Key; an optional argument used when the variable is included in a meter. The end use keys can be: 'GeneralLights', 'TaskLights', 'ExteriorLights', 'Heating', 'Cooling', 'DHW', 'Cogeneration', 'ExteriorEquipment', 'ZoneSource', 'PurchasedHotWater', 'PurchasedChilledWater', 'Fans', 'HeatingCoils', 'CoolingCoils', 'Pumps', 'Chillers', 'Boilers', 'Baseboard', 'HeatRejection', 'Humidifier', 'HeatRecovery', or 'Miscellaneous'.
GroupKey	Meter Super Group Key; an optional argument used when the variable is included in a meter. The group key denotes whether the variable belongs to the building, system, or plant. The choices are: 'Building', 'HVAC' or 'Plant'.

Table 3. SetupOutputVariable Arguments

As described in the *Input Output Reference*, not all variables may be available in any particular simulation. Only those variables that will have values generated will be

available for reporting. In the IDF, you can include a “Report Variable Dictionary” command that will produce the eplusout.rdd file containing all the variables with their IndexTypeKeys. This list can be used to tailor the requests for values in the OutputFileStandard.

Output Variable Dos and Don'ts

For general output variables there aren't many rules. For meter output variables there are quite a few. Here are some tips to keep you out of trouble.

What Variables Should I Output?

The choice of variables to output is really up to the developer. Since variables don't appear on the output file unless requested by the user in the IDF input file, it is better to “SetUp” too many rather than too few. For an HVAC component one should generally output the heating and cooling outputs of the component both in terms of energy and power. Energy is always output in Joules, power in Watts. If there is humidification or dehumidification both total and sensible cooling should be reported. Any electricity or fuel consumed by a component should be reported out, again both in terms of energy (Joules) and power (Watts). For HVAC components in most cases reporting inlet and outlet temperatures and humidities is unnecessary since these quantities can be obtained from the system node outputs.

Output Variable Naming Conventions

We have tried to obtain some consistency in variable names by defining some naming conventions. The heating and/or cooling output is always reported as:

```
<component-type> Heating Rate[W]
<component-type> Heating Energy[J]
<component-type> Total Cooling Rate[W]
<component-type> Total Cooling Energy[W]
<component-type> Sensible Cooling Rate[W]
<component-type> Sensible Cooling Energy[J]
```

Fuel and electricity consumption is reported as:

```
<component-type> Electric Power[W]
<component-type> Electric Consumption[J]
<component-type> Gas Consumption Rate[W]
<component-type> Gas Consumption[J]
```

Water addition is reported as:

```
<component-type> Water Consumption Rate[m3/s]
<component-type> Water Consumption[m3]
```

Units are always strictly SI and no abbreviations are allowed in the variable name. <component-type> is the type of component. It should not be the actual object class name from the IDD file, but rather one step of generality above this. For example for fancoils we have:

```
Fan Coil Total Cooling Energy[J]
```

Here <component-type> is “Fan Coil”, not FAN COIL UNIT:4 PIPE.

What are Meters?

In EnergyPlus meters are an additional output reporting capability. A meter is a way of grouping similar output variables. Meters are output variables just like ordinary output

variables except that they sum or average a collection of ordinary output variables. In EnergyPlus the meter variables serve two purposes.

1. Providing output of fuel and electricity consumption by end use categories and at the system plant, building and facility level.
2. Providing a way of summing heating or cooling outputs for a category of components. The resource type EnergyTransfer is used for this purpose. An example would be reporting out the sum of the heating energy from all the heating coils in a system.

How Do I Create A Meter?

Meter output variables are created at the same time and in the same manner as ordinary output variables. SetupOutputVariable is called but the optional arguments ResourceTypeKey, EndUseKey, and GroupKey must be used in addition to the usual arguments. For example, in the electric steam humidifier module

```
CALL SetupOutputVariable('Humidifier Electric Consumption[J]', &
  Humidifier(HumNum)%ElecUseEnergy, 'System','Sum', &
  Humidifier(HumNum)%Name)
```

creates an output variable labeled 'Humidifier Electric Consumption[J]' with the value of Humidifier(HumNum)%ElecUseEnergy.

```
CALL SetupOutputVariable('Humidifier Electric Consumption[J]', &
  Humidifier(HumNum)%ElecUseEnergy, 'System','Sum', &
  Humidifier(HumNum)%Name, &
  ResourceTypeKey='ELECTRICITY',EndUseKey = 'HUMIDIFIER', &
  GroupKey = 'System')
```

Creates the same output variable but in addition creates a meter output variable Humidifier:Electricity [J]. This variable will contain the sum of all the electricity consumption of the humidifiers in the system. In addition, this electrical consumption will be added into the meter variables Electricity:HVAC [J] and Electricity:Facility [J].


Rules for Meter Variables

There are a number of rules developers must follow in order to account for all electricity and fuel consumption as well as to prevent consumables from being double counted.

1. Electricity and fuel meters must always be defined at the simple component level. Some EnergyPlus components are compound components: they are built up from simple components. Examples are fan coils (composed of heating coils, cooling coils, and fans), terminal units etc. Some example simple components are heating and cooling coils, fans, humidifiers etc. Electricity and fuel consumption should always be metered at the simple component level and never at the compound component level. This prevents double counting of the fuel or energy consumption.
2. A variable should be metered once only. This means a variable can be assigned to only one resource type and to only one end use category.
3. Energy Transfer should be metered in the same way as fuel or electricity use. Energy Transfer meters should only be defined for simple components and should be assigned the same end use category as the fuel or electricity consumption.
4. All fuel and electricity consumption must be put in some (one) meter.

5. Use Energy Transfer judiciously; if in doubt, leave it out.

Important Rules for Module Developers

1. INITIALIZE!!!! INITIALIZE either fully or "invalidly" when you ALLOCATE the array/derived type. Two items have been set up to help you: BigNumber and DBigNumber are in DataGlobals. They get initialized before anything happens in the main routine (EnergyPlus). An invalid initialization will use one of these, appropriately.  don't know yet if this will trigger an exception when used...[DBS3]
2. Warning errors during "get input" should only be used when program termination is not required (this is rare). Each GetInput routine should be structured so that errors detected (such as an invalid schedule name which currently is just a warning) cause a fatal error after all the input for that item/module/etc is gotten. (See HBManager, BaseboardRadiator, others) In addition, don't make GetInputFlag a module variable. Make it as "local" as possible. Look at BaseboardRadiator for an example.
3. Error messages during simulation should be semi-intelligent. No one wants to see 5,000 messages saying "this flow invalid". If the error condition might happen a lot (especially during debugging), count each occurrence and only put out a message every 50 or so. (See example above in the Error Messages section). Also, if you are putting the same message in two modules, identify the error message with some designation. For Example, CALL ShowWarningError('SimRoutinename: this condition happened again') will help everyone track it down. Use the ShowContinueErrorTimeStamp so the time/date/environment of occurrence is known.
4. Use the templates for documentation! Modules, subroutines, functions templates all have been checked into StarTeam. Use them. Put INTENTS on your Subroutine Arguments. Document variables.
5. **Avoid the use of string comparisons in subroutines other than GetInput. Check string comparisons in the GetInput subroutines and assign a parameter for comparisons elsewhere in the module.**
6. **If you are submitting code to be placed into the public version of EnergyPlus, make sure that the proper "Grant-Back" procedure has been followed so that the correct attributions of code authorship are given as well as permission to use this code in publicly available software is assured. (see Appendix E, Code/Module Contribution Questionnaire)**

Appendix A. DataGlobals Module

```

MODULE DataGlobals      ! EnergyPlus Data-Only Module

! MODULE INFORMATION:
!   AUTHOR      Rick Strand
!   DATE WRITTEN January 1997
!   MODIFIED    May 1997 (RKS) Added Weather Variables
!   MODIFIED    December 1997 (RKS,DF,LKL) Split into DataGlobals and
DataEnvironment
!   MODIFIED    February 1999 (FW) Added NextHour, WGTNEXT, WGTNOW
!   MODIFIED    September 1999 (LKL) Rename WGTNEXT,WGTNOW for clarity
!   RE-ENGINEERED na

! PURPOSE OF THIS MODULE:
! This data-only module is a repository for all variables which are considered
! to be "global" in nature in EnergyPlus.

! METHODOLOGY EMPLOYED:
! na

! REFERENCES:
! na

! OTHER NOTES:
! na

! USE STATEMENTS:
! None!--This module is USED by all other modules; it should not USE anything.

IMPLICIT NONE      ! Enforce explicit typing of all variables

PUBLIC             ! By definition, all variables which are placed in this data
                  ! -only module should be available to other modules and routines.
                  ! Thus, all variables in this module must be PUBLIC.

! MODULE PARAMETER DEFINITIONS:
INTEGER, PARAMETER :: BeginDay = 1
INTEGER, PARAMETER :: DuringDay = 2
INTEGER, PARAMETER :: EndDay = 3
INTEGER, PARAMETER :: EndZoneSizingCalc = 4
INTEGER, PARAMETER :: EndSysSizingCalc = 5

INTEGER, PARAMETER :: ZoneTSReporting=1 ! value for Zone Time Step Reporting
(UpdateDataAndReport)
INTEGER, PARAMETER :: HVACTSReporting=2 ! value for HVAC Time Step Reporting
(UpdateDataAndReport)

REAL, PARAMETER    :: PI= 3.141592653589793 ! Pi
REAL, PARAMETER    :: PiOvr2 = PI/2.      ! Pi/2
REAL, PARAMETER    :: DegToRadians = PI/180. ! Conversion for Degrees to Radians
REAL, PARAMETER    :: SecInHour = 3600.0    ! Conversion for hours to seconds
INTEGER, PARAMETER :: MaxNameLength = 60     ! Maximum Name Length in Characters -- should be the
same
                                           ! as MaxAlphaArgLength in InputProcessor module

REAL, PARAMETER    :: InitConvTemp = 5.05    ! [deg C], standard init vol to mass flow conversion
temp
INTEGER, PARAMETER :: MaxSlatAngs = 19
INTEGER, PARAMETER :: MaxRefPoints = 102     ! Maximum number of daylighting reference points, 2
+ 10*10

! DERIVED TYPE DEFINITIONS:
! na

! INTERFACE BLOCK SPECIFICATIONS:
INTERFACE

```



```

SUBROUTINE ShowMessage(Message,Unit1,Unit2)
! Use when you want to create your own message for the error file.
CHARACTER(len=*) Message ! Message automatically written to "error file"
INTEGER, OPTIONAL :: Unit1 ! Unit number of open formatted file for message
INTEGER, OPTIONAL :: Unit2 ! Unit number of open formatted file for message
END SUBROUTINE
END INTERFACE
INTERFACE
SUBROUTINE ShowContinueError(Message,Unit1,Unit2)
! Use when you are "continuing" an error message over several lines.
CHARACTER(len=*) Message ! Message automatically written to "error file"
INTEGER, OPTIONAL :: Unit1 ! Unit number of open formatted file for message
INTEGER, OPTIONAL :: Unit2 ! Unit number of open formatted file for message
END SUBROUTINE
END INTERFACE
INTERFACE
SUBROUTINE ShowContinueErrorTimeStamp(Message,Unit1,Unit2)
! Use when you are "continuing" an error message and want to show the environment, day and
time.
CHARACTER(len=*) Message ! Message automatically written to "error file"
INTEGER, OPTIONAL :: Unit1 ! Unit number of open formatted file for message
INTEGER, OPTIONAL :: Unit2 ! Unit number of open formatted file for message
END SUBROUTINE
END INTERFACE
INTERFACE
SUBROUTINE ShowFatalError(Message,Unit1,Unit2)
! Use when you want the program to terminate after writing messages
! to appropriate files
CHARACTER(len=*) Message ! Message automatically written to "error file"
INTEGER, OPTIONAL :: Unit1 ! Unit number of open formatted file for message
INTEGER, OPTIONAL :: Unit2 ! Unit number of open formatted file for message
END SUBROUTINE
END INTERFACE
INTERFACE
SUBROUTINE ShowSevereError(Message,Unit1,Unit2)
! Use for "severe" error messages. Might have several severe tests and then terminate.
CHARACTER(len=*) Message ! Message automatically written to "error file"
INTEGER, OPTIONAL :: Unit1 ! Unit number of open formatted file for message
INTEGER, OPTIONAL :: Unit2 ! Unit number of open formatted file for message
END SUBROUTINE
END INTERFACE
INTERFACE
SUBROUTINE ShowWarningError(Message,Unit1,Unit2)
! Use for "warning" error messages.
CHARACTER(len=*) Message ! Message automatically written to "error file"
INTEGER, OPTIONAL :: Unit1 ! Unit number of open formatted file for message
INTEGER, OPTIONAL :: Unit2 ! Unit number of open formatted file for message
END SUBROUTINE
END INTERFACE

INTERFACE SetupOutputVariable
SUBROUTINE
SetupRealOutputVariable(VariableName,ActualVariable,IndexTypeKey,VariableTypeKey,KeyedValue, &
ReportFreq,ResourceTypeKey,EndUseKey,GroupKey)
CHARACTER(len=*), INTENT(IN) :: VariableName ! String Name of variable
REAL, INTENT(IN), TARGET :: ActualVariable ! Actual Variable, used to set up pointer
CHARACTER(len=*), INTENT(IN) :: IndexTypeKey ! Zone, HeatBalance=1, HVAC, System, Plant=2
CHARACTER(len=*), INTENT(IN) :: VariableTypeKey ! State, Average=1, NonState, Sum=2
CHARACTER(len=*), INTENT(IN) :: KeyedValue ! Associated Key for this variable
CHARACTER(len=*), INTENT(IN), OPTIONAL :: ReportFreq ! Internal use -- causes reporting
at this frequency
CHARACTER(len=*), INTENT(IN), OPTIONAL :: ResourceTypeKey ! Meter Resource Type
(Electricity, Gas, etc)
CHARACTER(len=*), INTENT(IN), OPTIONAL :: EndUseKey ! Meter End Use Key (Task Lights,
Heating, Cooling, etc)
CHARACTER(len=*), INTENT(IN), OPTIONAL :: GroupKey ! Meter Super Group Key (Building,
System, Plant)
END SUBROUTINE
SUBROUTINE
SetupIntegerOutputVariable(VariableName,IntActualVariable,IndexTypeKey,VariableTypeKey,KeyedValue,
ReportFreq)

```

```

    CHARACTER(len=*), INTENT(IN) :: VariableName ! String Name of variable
    INTEGER, INTENT(IN), TARGET :: IntActualVariable ! Actual Variable, used to set up pointer
    CHARACTER(len=*), INTENT(IN) :: IndexTypeKey ! Zone, HeatBalance=1, HVAC, System, Plant=2
    CHARACTER(len=*), INTENT(IN) :: VariableTypeKey ! State, Average=1, NonState, Sum=2
    CHARACTER(len=*), INTENT(IN) :: KeyedValue ! Associated Key for this variable
    CHARACTER(len=*), INTENT(IN), OPTIONAL :: ReportFreq ! Internal use -- causes reporting
at this frequency
    END SUBROUTINE
    SUBROUTINE
SetupRealOutputVariable_IntKey(VariableName,ActualVariable,IndexTypeKey,VariableTypeKey,KeyedValue
, &
    ReportFreq,ResourceTypeKey,EndUseKey,GroupKey)
    CHARACTER(len=*), INTENT(IN) :: VariableName ! String Name of variable
    REAL, INTENT(IN), TARGET :: ActualVariable ! Actual Variable, used to set up pointer
    CHARACTER(len=*), INTENT(IN) :: IndexTypeKey ! Zone, HeatBalance=1, HVAC, System, Plant=2
    CHARACTER(len=*), INTENT(IN) :: VariableTypeKey ! State, Average=1, NonState, Sum=2
    INTEGER, INTENT(IN) :: KeyedValue ! Associated Key for this variable
    CHARACTER(len=*), INTENT(IN), OPTIONAL :: ReportFreq ! Internal use -- causes reporting
at this frequency
    CHARACTER(len=*), INTENT(IN), OPTIONAL :: ResourceTypeKey ! Meter Resource Type
(Electricity, Gas, etc)
    CHARACTER(len=*), INTENT(IN), OPTIONAL :: EndUseKey ! Meter End Use Key (Task Lights,
Heating, Cooling, etc)
    CHARACTER(len=*), INTENT(IN), OPTIONAL :: GroupKey ! Meter Super Group Key (Building,
System, Plant)
    END SUBROUTINE
    END INTERFACE

    INTERFACE SetupRealInternalOutputVariable
    INTEGER FUNCTION
SetupRealInternalOutputVariable(VariableName,ActualVariable,IndexTypeKey,VariableTypeKey, &
    KeyedValue,ReportFreq)
    CHARACTER(len=*), INTENT(IN) :: VariableName ! String Name of variable
    CHARACTER(len=*), INTENT(IN) :: IndexTypeKey ! Zone, HeatBalance or HVAC, System, Plant
    CHARACTER(len=*), INTENT(IN) :: VariableTypeKey ! State, Average, or NonState, Sum
    REAL, INTENT(IN), TARGET :: ActualVariable ! Actual Variable, used to set up pointer
    CHARACTER(len=*), INTENT(IN) :: KeyedValue ! Associated Key for this variable
    CHARACTER(len=*), INTENT(IN) :: ReportFreq ! Frequency to store
'timestep','hourly','monthly','environment'
    END FUNCTION
    END INTERFACE

    INTERFACE GetInternalVariableValue
    REAL FUNCTION GetInternalVariableValue(WhichVar)
    INTEGER, INTENT(IN) :: WhichVar ! Report number assigned to this variable
    END FUNCTION
    END INTERFACE

    ! MODULE VARIABLE DECLARATIONS:

LOGICAL :: BeginDayFlag ! true at the start of each day, false after first time step in
day
LOGICAL :: BeginEnvrnFlag ! true at the start of each environment, false after first time
step in environ
LOGICAL :: BeginHourFlag ! true at the start of each hour, false after first time step in
hour
LOGICAL :: BeginSimFlag ! true until any actual simulation (full or sizing) has begun,
! false after first time step
LOGICAL :: BeginFullSimFlag ! true until full simulation has begun,
! false after first time step
LOGICAL :: BeginTimeStepFlag ! true at the start of each time step, false after first subtime
step of time step
REAL :: BigNumber ! Max Number (real) used for initializations
DOUBLE PRECISION :: DBigNumber ! Max Number (double precision) used for initializations
INTEGER :: DayOfSim ! Counter for days (during the simulation)
CHARACTER(len=25) :: DayOfSimChr ! Counter for days (during the simulation) (character -- for
reporting, large so can use write *)
LOGICAL :: EndEnvrnFlag ! true at the end of each environment (last time step of last hour
of last day of environ)
LOGICAL :: EndDayFlag ! true at the end of each day (last time step of last hour of day)
LOGICAL :: EndHourFlag ! true at the end of each hour (last time step of hour)

```

```

INTEGER :: PreviousHour          ! Previous Hour Index
INTEGER :: HourOfDay             ! Counter for hours in a simulation day
DOUBLE PRECISION :: WeightPreviousHour ! Weighting of value for previous hour
DOUBLE PRECISION :: WeightNow    ! Weighting of value for current hour
INTEGER :: NumOfDayInEnvrn       ! Number of days in the simulation for a particular environment
INTEGER :: NumOfTimeStepInHour   ! Number of time steps in each hour of the simulation
INTEGER :: NumOfZones            ! Total number of Zones for simulation
INTEGER :: TimeStep              ! Counter for time steps (fractional hours)
REAL    :: TimeStepZone          ! Zone time step in fractional hours
LOGICAL :: WarmupFlag            ! true during the warmup portion of a simulation
INTEGER :: OutputFileStandard    ! Unit number for the standard output file (hourly data only)
INTEGER :: StdOutputRecordCount  ! Count of Standard output records
INTEGER :: OutputFileInits       ! Unit number for the standard Initialization output file
INTEGER :: OutputFileDebug       ! Unit number for debug outputs
CHARACTER(len=1) :: SizingFileColSep=' ' ! Character to separate columns in sizing outputs
INTEGER :: OutputFileZoneSizing ! Unit number of zone sizing calc output file
INTEGER :: OutputFileSysSizing  ! Unit number of system sizing calc output file
INTEGER :: OutputFileMeters     ! Unit number for meters output
INTEGER :: StdMeterRecordCount  ! Count of Meter output records
INTEGER :: OutputFileBNDetails  ! Unit number for Branch-Node Details
INTEGER :: OutputFileConstrainParams ! Unit number for special constrained free parameters output
                                     ! file (for penalty functions in optimizing)

LOGICAL :: DebugOutput
LOGICAL :: EvenDuringWarmup
LOGICAL :: ZoneSizingCalc = .FALSE. ! TRUE if zone sizing calculation
LOGICAL :: SysSizingCalc = .FALSE.  ! TRUE if system sizing calculation
LOGICAL :: DoZoneSizing           ! User input in RUN CONTROL object
LOGICAL :: DoSystemSizing         ! User input in RUN CONTROL object
LOGICAL :: DoPlantSizing          ! User input in RUN CONTROL object
LOGICAL :: DoDesDaySim            ! User input in RUN CONTROL object
LOGICAL :: DoWeathSim             ! User input in RUN CONTROL object
LOGICAL :: WeathSimReq = .false. ! Input has a RunPeriod request
LOGICAL :: WeatherFile           ! TRUE if current environment is a weather file
LOGICAL :: DoOutputReporting      ! TRUE if variables to be written out
LOGICAL :: DoingSizing=.false. ! TRUE when "sizing" is being performed (some error messages won't
be displayed)
LOGICAL :: DisplayFluidPropsErrors=.true. ! false when Fluid Properties Warnings should not be
displayed (e.g. during warmup)
DOUBLE PRECISION :: CurrentTime ! CurrentTime, in fractional hours, from start of day. Uses Loads
time step.
INTEGER :: SimTimeSteps          ! Number of (Loads) timesteps since beginning of run period
(environment).
INTEGER :: MinutesPerTimeStep    ! Minutes per time step calculated from NumTimeStepInHour (number
of minutes per load time step)
LOGICAL :: DisplayPerfSimulationFlag ! true when "Performing Simulation" should be displayed

!
!   NOTICE
!
!   Copyright © 1996-2004 The Board of Trustees of the University of Illinois
!   and The Regents of the University of California through Ernest Orlando Lawrence
!   Berkeley National Laboratory. All rights reserved.
!
!   Portions of the EnergyPlus software package have been developed and copyrighted
!   by other individuals, companies and institutions. These portions have been
!   incorporated into the EnergyPlus software package under license. For a complete
!   list of contributors, see "Notice" located in EnergyPlus.f90.
!
!   NOTICE: The U.S. Government is granted for itself and others acting on its
!   behalf a paid-up, nonexclusive, irrevocable, worldwide license in this data to
!   reproduce, prepare derivative works, and perform publicly and display publicly.
!   Beginning five (5) years after permission to assert copyright is granted,
!   subject to two possible five year renewals, the U.S. Government is granted for
!   itself and others acting on its behalf a paid-up, non-exclusive, irrevocable
!   worldwide license in this data to reproduce, prepare derivative works,
!   distribute copies to the public, perform publicly and display publicly, and to
!   permit others to do so.
!
!   TRADEMARKS: EnergyPlus is a trademark of the US Department of Energy.
!
END MODULE DataGlobals

```

Appendix B. DataEnvironment Module

```

MODULE DataEnvironment      ! EnergyPlus Data-Only Module

! MODULE INFORMATION:
!   AUTHOR      Rick Strand, Dan Fisher, Linda Lawrie
!   DATE WRITTEN December 1997
!   MODIFIED    November 1998, Fred Winkelmann
!   MODIFIED    June 1999, June 2000, Linda Lawrie
!   RE-ENGINEERED na

! PURPOSE OF THIS MODULE:
! This data-only module is a repository for the variables that relate specifically
! to the "environment" (i.e. current date data, tomorrow's date data, and
! current weather variables)

! METHODOLOGY EMPLOYED:
! na

! REFERENCES:
! na

! OTHER NOTES:
! na

! USE STATEMENTS:
USE DataGlobals, ONLY: MaxNameLength

IMPLICIT NONE      ! Enforce explicit typing of all variables

PUBLIC             ! By definition, all variables which are placed in this data
                  ! -only module should be available to other modules and routines.
                  ! Thus, all variables in this module must be PUBLIC.

! MODULE PARAMETER DEFINITIONS:
! na

! DERIVED TYPE DEFINITIONS
! na

! INTERFACE BLOCK SPECIFICATIONS
! na

! MODULE VARIABLE DECLARATIONS:

REAL      :: BeamSolarRad           ! Current beam normal solar irradiance
INTEGER   :: DayOfMonth             ! Current day of the month
INTEGER   :: DayOfMonthTomorrow     ! Tomorrow's day of the month
INTEGER   :: DayOfWeek              ! Current day of the week (Sunday=1, Monday=2, ...)
INTEGER   :: DayOfWeekTomorrow      ! Tomorrow's day of the week (Sunday=1, Monday=2, ...)
INTEGER   :: DayOfYear              ! Current day of the year (01JAN=1, 02JAN=2, ...)
REAL      :: DifSolarRad            ! Current sky diffuse solar horizontal irradiance
INTEGER   :: DSTIndicator           ! Daylight Saving Time Indicator (1=yes, 0=no) for Today
REAL      :: Elevation              ! Elevation of this building site
LOGICAL   :: EndMonthFlag           ! Set to true on last day of month
REAL      :: GndReflectanceForDayltg ! Ground visible reflectance for use in daylighting calc
REAL      :: GndReflectance         ! Ground visible reflectance from input
REAL      :: GndSolarRad            ! Current ground reflected radiation
REAL      :: GroundTemp             ! Current ground temperature
REAL      :: GroundTemp_Surface     ! Current surface ground temperature
REAL      :: GroundTemp_Deep        ! Current deep ground temperature
INTEGER   :: HolidayIndex           ! Indicates whether current day is a holiday and if so what
type                                           type
                                           ! HolidayIndex=(0-no holiday, 1-holiday type 1, ...)
INTEGER   :: HolidayIndexTomorrow   ! Tomorrow's Holiday Index
LOGICAL   :: IsRain                 ! Surfaces are wet for this time interval
LOGICAL   :: IsSnow                 ! Snow on the ground for this time interval

```

```

REAL      :: Latitude           ! Latitude of building location
REAL      :: Longitude         ! Longitude of building location
INTEGER   :: Month             ! Current calendar month
INTEGER   :: MonthTomorrow     ! Tomorrow's calendar month
REAL      :: OutBaroPress      ! Current outdoor air barometric pressure
REAL      :: OutDryBulbTemp    ! Current outdoor air dry bulb temperature
REAL      :: OutHumRat         ! Current outdoor air humidity ratio
REAL      :: OutRelHum         ! Current outdoor relative humidity [%]
REAL      :: OutRelHumValue    ! Current outdoor relative humidity value [0.0-1.0]
REAL      :: OutEnthalpy       ! Current outdoor enthalpy
REAL      :: OutAirDensity     ! Current outdoor air density
REAL      :: OutWetBulbTemp    ! Current outdoor air wet bulb temperature
REAL      :: OutDewPointTemp   ! Current outdoor dewpoint temperature
REAL      :: SkyTemp           ! Current sky temperature
LOGICAL   :: SunIsUp           ! True when Sun is over horizon, False when not
REAL      :: WindDir           ! Current outdoor air wind direction
REAL      :: WindSpeed         ! Current outdoor air wind speed
INTEGER   :: Year              ! Current calendar year of the simulation
INTEGER   :: YearTomorrow     ! Tomorrow's calendar year of the simulation
DOUBLE PRECISION, DIMENSION(3) :: SOLCOS      ! Solar direction cosines at current time step
REAL      :: CloudFraction     ! Fraction of sky covered by clouds
REAL      :: HISKF             ! Exterior horizontal illuminance from sky (lux).
REAL      :: HISUNF            ! Exterior horizontal beam illuminance (lux)
REAL      :: HISUNFnorm        ! Exterior beam normal illuminance (lux)
REAL      :: PDIRLW            ! Luminous efficacy (lum/W) of beam solar radiation
REAL      :: PDIFLW            ! Luminous efficacy (lum/W) of sky diffuse solar radiation
REAL      :: SkyClearness      ! Sky clearness (see subr. DayltgLuminousEfficacy)
REAL      :: SkyBrightness     ! Sky brightness (see subr. DayltgLuminousEfficacy)
REAL      :: StdBaroPress      ! Standard "atmospheric pressure" based on elevation (ASHRAE
HOF p6.1)
REAL      :: TimeZoneNumber    ! Time Zone Number of building location
REAL      :: TimeZoneMeridian  ! Standard Meridian of TimeZone
CHARACTER(len=100) :: EnvironmentName ! Current environment name (longer for weather file names)
CHARACTER(len=20)  :: CurMnDyHr ! Current Month/Day/Hour timestamp info
CHARACTER(len=5)   :: CurMnDy   ! Current Month/Day timestamp info
INTEGER   :: CurEnvirNum        ! current environment number
Integer   :: TotDesDays         ! Total number of Design days to Setup
INTEGER   :: CurrentOverallSimDay ! Count of current simulation day in total of all sim days
INTEGER   :: TotalOverallSimDays ! Count of all possible simulation days in all environments
INTEGER   :: MaxNumberSimYears  ! Maximum number of simulation years requested in all RunPeriod
statements

DOUBLE PRECISION :: CosSolarDeclinAngle ! Cosine of the solar declination angle
DOUBLE PRECISION :: EquationOfTime      ! Value of the equation of time formula
DOUBLE PRECISION :: SinLatitude         ! Sine of Latitude
DOUBLE PRECISION :: CosLatitude         ! Cosine of Latitude
DOUBLE PRECISION :: SinSolarDeclinAngle ! Sine of the solar declination angle

LOGICAL :: GroundTempObjInput=.false. ! Ground temperature object input
LOGICAL :: GroundTemp_SurfaceObjInput=.false. ! Surface ground temperature object input
LOGICAL :: GroundTemp_DeepObjInput=.false. ! Deep ground temperature object input

!
!   NOTICE
!
!   Copyright © 1996-2004 The Board of Trustees of the University of Illinois
!   and The Regents of the University of California through Ernest Orlando Lawrence
!   Berkeley National Laboratory. All rights reserved.
!
!   Portions of the EnergyPlus software package have been developed and copyrighted
!   by other individuals, companies and institutions. These portions have been
!   incorporated into the EnergyPlus software package under license. For a complete
!   list of contributors, see "Notice" located in EnergyPlus.f90.
!
!   NOTICE: The U.S. Government is granted for itself and others acting on its
!   behalf a paid-up, nonexclusive, irrevocable, worldwide license in this data to
!   reproduce, prepare derivative works, and perform publicly and display publicly.
!   Beginning five (5) years after permission to assert copyright is granted,
!   subject to two possible five year renewals, the U.S. Government is granted for
!   itself and others acting on its behalf a paid-up, non-exclusive, irrevocable
!   worldwide license in this data to reproduce, prepare derivative works,
!   distribute copies to the public, perform publicly and display publicly, and to

```

```
!      permit others to do so.  
!  
!      TRADEMARKS: EnergyPlus is a trademark of the US Department of Energy.  
!  
END MODULE DataEnvironment
```

Appendix C. Submissions and Check-ins

There are two methods by which new modules are entered into the EnergyPlus (publicly available) program.

- **Checkin:** Part of the core development team may create or modify an existing module. As we use a configuration management system – this is called a check in.
- **Submission:** When someone outside the core development team submits a module or modification of an existing module for inclusion, this is termed a submission.

Both kinds of inclusions need to follow the check list procedure for new inclusions:

✓ **Source Code Rules**

Shall follow programming standard

Shall follow F90/95 standards

Shall follow the Template standards (documentation, naming conventions)

Shall follow the guidelines shown in this document

No Tabs in source code!!!

Lines shall be less than 133 characters in length. (Some compilers allow longer lines without warning).

Use F95 standards checking during compiles -- you may use the compiler option to generate warnings for non-standard code.

Permission to use the code must be supplied -- written, even email, is required. LBNL is monitoring this aspect -- so a grant-back letter can also be obtained from them.

✓ **Energy+.IDD rules**

Standard Units shall be used (SI only on Input)

Show units with the `\units` field. Supply `\ip-units` only if your input would require it (see comments at top of the Energy+.idd).

Use `\minimum` and `\maximum`

The first field following the object name should contain “**name**” as part of the field name

Use `\default` and `\required-field` appropriately

Object changes during minor releases (x.x.xxx) should not change fields in the middle – only at the end

Surface objects may not add to the end (4 vertices are all that is currently allowed but we will be changing that!)

Note that significant changes in the Energy+.idd may require a “transition” rule change in the Rules Spreadsheet file (Rules...xls).

✓ **Testing**

Should run the test suite (full desired). We have a python script that can compare between two run versions (using the .csv files output from ReadVarsESO).

If you need a script, look under StarTeam...Test Files>ScriptMaker

If you modify objects, you are responsible for changing the test suite files that are impacted by your object modifications.

You should add a new input file for your changes—input files should be documented! (Test files have a document template as well).

You should run a full annual run with your test file even if that is not the configuration that ends up in the internal test suite.

✓ **Documentation (must be included at the same time as code!!!)**

A document template is available for use – only the styles in that document should be used. (Microsoft™ Word is our standard word processing software).

Equations – limited in IOREf, necessary in Engineering Doc – limit the number of “references” though. You can use standard Equation formatting from Microsoft™ Word.

Figures – Though AutoShapes may draw nice pictures, they are not often “captionable” without undue editing. Please make figures into Jpegs or GIFs. Use “insert caption” (below the figure) so that auto-numbering of figures is used (these will transfer automatically to EnergyPlus documents).

Tables – use “insert caption” (above the table) so that auto-numbering of figures is used (these will transfer automatically to EnergyPlus documents).

Cross-References – limit your “insert cross references”. You may highlight these so that “editing” from your inclusion is more obvious – use a different color to help them stand out.

IORef – See the InputOutputReference document for indications of what is included.

Eng Ref – Most new modules shall include an engineering document reference. See the Engineering Document for indications of typical writeups.

Output Details and Examples – this can help illustrate your changes.

✓ **FeatureChanges.csv**

Most changes should include a line in the “featurechanges.csv” file

✓ **Checked in?**

A courtesy message to the EnergyPlus team should be done for each check in, with details of files checked in, etc. Save one of the emails you have received if you don't know how many to send it to.

✓ **Defect fixing?**

If you fix a defect or “fix” a suggested change (CR), you should mark it “fixed” in StarTeam and the responsibility should automatically change back to the author of the CR.

✓ **Rules...xls**

If a transition rule will be needed (or a deleted / obsolete / renamed object is needed) – a line (or more) in this spreadsheet should be used. See example rules files from previous releases. If in doubt, put something in.

✓ **ReportVariables...csv**

If you change the name of a report variable, the transition program for release can automatically transition older input files IF you put the old and new names into this file.

Appendix D. Documentation Specifics

Documents that module developers will typically be updating/changing are the: Input Output Reference, Engineering Documentation, and Output Details and Examples. You may, of course, note revisions to other documents.

All of the EnergyPlus documentation follows a Word™ template – report.dot.

This template takes care of many of the nuances of formatting so that the documents all retain the same “look and feel”. The template itself will contain examples for the IORef and Engineering Documentation.

General guidelines:

- Don't get fancy with formatting. No extra “enters” are needed to space the paragraphs.
- Submit your pictures as pictures (jpeg, tif, gif). This will allow you to “insert captions” below them and have them automatically numbered. (This also allows them to be re-numbered once inside the EnergyPlus documents).
- Likewise, use an “insert caption” on tables.
- Table captions go above the table. Figure captions go below.
- If you want to reference a table or figure in your text, use “insert cross reference” and select table or figure as appropriate. Usually, just use the “label and number” option.
- Body Text is the expected style for most text.
- Headings are used judiciously to help separate text.
- Object names (IOReference) are Heading 3.
- Each field should be described and shown as Heading 4 followed by the description.
- Each object's IDD should be shown and use the format “IDD Definition”.
- An excerpt IDF using the object should be shown.
- Output variables for the object should be shown (heading 4) with a heading 3 <object name> Output variables preceding.
- Equations may be inserted using the Microsoft™ Equation Editor. Internally we use software called “MathType” – that also may be used for Equations. It is not desirable to number every equation. If you want to reference the equations, of course, you will need to number them – it is best to number them in plain text and then we can edit them into the rest of the documents.
- Each Engineering Reference section should contain a “References” section and should be formatted in author style (not numbered).

Appendix E. Module Template

The following module template can and should be used to create new modules. Following the module template are subroutine and function templates. You should be able to copy the template for your own use (or you can get a plain text version).

```

MODULE <module_name>

! Module containing the routines dealing with the <module_name>

! MODULE INFORMATION:
!   AUTHOR      <author>
!   DATE WRITTEN <date_written>
!   MODIFIED    na
!   RE-ENGINEERED na

! PURPOSE OF THIS MODULE:
! Needs description

! METHODOLOGY EMPLOYED:
! Needs description, as appropriate.

! REFERENCES: none

! OTHER NOTES: none

! USE STATEMENTS:
! Use statements for data only modules
USE DataGlobals, ONLY: ShowWarningError, ShowSevereError, ShowFatalError, &
                        MaxNameLength, ...

! Use statements for access to subroutines in other modules

IMPLICIT NONE          ! Enforce explicit typing of all variables

PRIVATE ! Everything private unless explicitly made public

! MODULE PARAMETER DEFINITIONS
! na

! DERIVED TYPE DEFINITIONS

! MODULE VARIABLE DECLARATIONS:

! SUBROUTINE SPECIFICATIONS FOR MODULE <module_name>

! Name Public routines, optionally name Private routines within this module

PUBLIC Sim<module_name>
PRIVATE Get<module_name>
PRIVATE Calc<module_name>
PRIVATE Update<module_name>
PRIVATE Report<module_name>

CONTAINS

```

```

SUBROUTINE Sim<module_name>

    ! SUBROUTINE INFORMATION:
    !     AUTHOR             <author>
    !     DATE WRITTEN       <date_written>
    !     MODIFIED           na
    !     RE-ENGINEERED      na

    ! PURPOSE OF THIS SUBROUTINE:
    ! This subroutine needs a description.

    ! METHODOLOGY EMPLOYED:
    ! Needs description, as appropriate.

    ! REFERENCES:
    ! na

    ! USE STATEMENTS:
    ! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

    ! SUBROUTINE ARGUMENT DEFINITIONS:
    ! na

    ! SUBROUTINE PARAMETER DEFINITIONS:
    ! na

    ! INTERFACE BLOCK SPECIFICATIONS
    ! na

    ! DERIVED TYPE DEFINITIONS
    ! na

    ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
    LOGICAL,SAVE      :: GetInputFlag = .true.  ! First time, input is "gotten"

    IF (GetInputFlag) THEN
        CALL Get<module_name>Input
        GetInputFlag=.false.
    ENDIF

    <... insert any necessary code here>

    CALL Init<module_name>(Args)

    CALL Calc<module_name>(Args)

    CALL Update<module_name>(Args)

    CALL Report<module_name>(Args)

    RETURN

END SUBROUTINE Sim<module_name>

SUBROUTINE Get<module_name>Input

    ! SUBROUTINE INFORMATION:
    !     AUTHOR             <author>
    !     DATE WRITTEN       <date_written>
    !     MODIFIED           na
    !     RE-ENGINEERED      na

    ! PURPOSE OF THIS SUBROUTINE:
    ! This subroutine needs a description.

    ! METHODOLOGY EMPLOYED:
    ! Needs description, as appropriate.

    ! REFERENCES:

```

```

! na

! USE STATEMENTS:
USE InputProcessor, ONLY: GetNumObjectsFound, GetObjectItem ! might also use FindItemInList

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

! SUBROUTINE ARGUMENT DEFINITIONS:
! na

! SUBROUTINE PARAMETER DEFINITIONS:
! na

! INTERFACE BLOCK SPECIFICATIONS
! na

! DERIVED TYPE DEFINITIONS
! na

! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
INTEGER                :: Item      ! Item to be "gotten"
CHARACTER(len=MaxNameLength), &
    DIMENSION(x) :: Alphas ! Alpha items for object
REAL, DIMENSION(y)   :: Numbers ! Numeric items for object
INTEGER              :: NumAlphas ! Number of Alphas for each GetObjectItem call
INTEGER              :: NumNumbers ! Number of Numbers for each GetObjectItem call
INTEGER              :: IOSStatus  ! Used in GetObjectItem
LOGICAL              :: ErrorsFound=.false. ! Set to true if errors in input, fatal
at end of routine

<NumItems>=GetNumObjectsFound('object for <module_name>')
DO Item=1,<NumItems>
    CALL GetObjectItem('object for
<module_name>',Item,Alphas,NumAlphas,Numbers,NumNumbers,IOSStatus)
    <process, noting errors>
ENDDO

<SetupOutputVariables here...>

IF (ErrorsFound) THEN
    CALL ShowFatalError('Get<module_name>Input: Errors found in input')
ENDIF

RETURN
END SUBROUTINE Get<module_name>Input

SUBROUTINE Init<module_name>

! SUBROUTINE INFORMATION:
!     AUTHOR          <author>
!     DATE WRITTEN    <date_written>
!     MODIFIED        na
!     RE-ENGINEERED   na

! PURPOSE OF THIS SUBROUTINE:
! This subroutine needs a description.

! METHODOLOGY EMPLOYED:
! Needs description, as appropriate.

! REFERENCES:
! na

! USE STATEMENTS:
! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

! SUBROUTINE ARGUMENT DEFINITIONS:
! na

```

```

        ! SUBROUTINE PARAMETER DEFINITIONS:
        ! na

        ! INTERFACE BLOCK SPECIFICATIONS
        ! na

        ! DERIVED TYPE DEFINITIONS
        ! na

        ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
        ! na

RETURN
END SUBROUTINE Init<module_name>

SUBROUTINE Size<module_name>

        ! SUBROUTINE INFORMATION:
        !       AUTHOR      <author>
        !       DATE WRITTEN <date_written>
        !       MODIFIED     na
        !       RE-ENGINEERED na

        ! PURPOSE OF THIS SUBROUTINE:
        ! This subroutine needs a description.

        ! METHODOLOGY EMPLOYED:
        ! Needs description, as appropriate.

        ! REFERENCES:
        ! na

        ! USE STATEMENTS:
        ! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

        ! SUBROUTINE ARGUMENT DEFINITIONS:
        ! na

        ! SUBROUTINE PARAMETER DEFINITIONS:
        ! na

        ! INTERFACE BLOCK SPECIFICATIONS
        ! na

        ! DERIVED TYPE DEFINITIONS
        ! na

        ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
        ! na

RETURN
END SUBROUTINE Size<module_name>

SUBROUTINE Calc<module_name>

        ! SUBROUTINE INFORMATION:
        !       AUTHOR      <author>
        !       DATE WRITTEN <date_written>
        !       MODIFIED     na
        !       RE-ENGINEERED na

        ! PURPOSE OF THIS SUBROUTINE:
        ! This subroutine needs a description.

        ! METHODOLOGY EMPLOYED:
        ! Needs description, as appropriate.

```

```

! REFERENCES:
! na

! USE STATEMENTS:
! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

! SUBROUTINE ARGUMENT DEFINITIONS:
! na

! SUBROUTINE PARAMETER DEFINITIONS:
! na

! INTERFACE BLOCK SPECIFICATIONS
! na

! DERIVED TYPE DEFINITIONS
! na

! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
! na

RETURN

END SUBROUTINE Calc<module_name>

SUBROUTINE Update<module_name>

! SUBROUTINE INFORMATION:
!   AUTHOR          <author>
!   DATE WRITTEN    <date_written>
!   MODIFIED        na
!   RE-ENGINEERED   na

! PURPOSE OF THIS SUBROUTINE:
! This subroutine needs a description.

! METHODOLOGY EMPLOYED:
! Needs description, as appropriate.

! REFERENCES:
! na

! USE STATEMENTS:
! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

! SUBROUTINE ARGUMENT DEFINITIONS:
! na

! SUBROUTINE PARAMETER DEFINITIONS:
! na

! INTERFACE BLOCK SPECIFICATIONS
! na

! DERIVED TYPE DEFINITIONS
! na

! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
! na

RETURN

END SUBROUTINE Update<module_name>

SUBROUTINE Report<module_name>

```

```

! SUBROUTINE INFORMATION:
!   AUTHOR           <author>
!   DATE WRITTEN     <date_written>
!   MODIFIED         na
!   RE-ENGINEERED    na

! PURPOSE OF THIS SUBROUTINE:
! This subroutine needs a description.

! METHODOLOGY EMPLOYED:
! Needs description, as appropriate.

! REFERENCES:
! na

! USE STATEMENTS:
! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

! SUBROUTINE ARGUMENT DEFINITIONS:
! na

! SUBROUTINE PARAMETER DEFINITIONS:
! na

! INTERFACE BLOCK SPECIFICATIONS
! na

! DERIVED TYPE DEFINITIONS
! na

! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
! na

< this routine is typically needed only for those cases where you must transform the internal
data to a reportable form>

RETURN

END SUBROUTINE Report<module_name>

END MODULE <module_name>

```

The Subroutine Template:

```

SUBROUTINE <name>

! SUBROUTINE INFORMATION:
!   AUTHOR           <author>
!   DATE WRITTEN     <date_written>
!   MODIFIED         na
!   RE-ENGINEERED    na

! PURPOSE OF THIS SUBROUTINE:
! This subroutine needs a description.

! METHODOLOGY EMPLOYED:
! Needs description, as appropriate.

! REFERENCES:
! na

! USE STATEMENTS:
! na

```



```

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

      ! SUBROUTINE ARGUMENT DEFINITIONS:
      ! na

      ! SUBROUTINE PARAMETER DEFINITIONS:
      ! na

      ! INTERFACE BLOCK SPECIFICATIONS
      ! na

      ! DERIVED TYPE DEFINITIONS
      ! na

      ! SUBROUTINE LOCAL VARIABLE DECLARATIONS:
      ! na

RETURN

END SUBROUTINE <name>

```

And the Function Template:

```

<type> FUNCTION <name>

      ! FUNCTION INFORMATION:
      !       AUTHOR      <author>
      !       DATE WRITTEN <date_written>
      !       MODIFIED     na
      !       RE-ENGINEERED na

      ! PURPOSE OF THIS FUNCTION:
      ! This function needs a description.

      ! METHODOLOGY EMPLOYED:
      ! Needs description, as appropriate.

      ! REFERENCES:
      ! na

      ! USE STATEMENTS:
      ! na

IMPLICIT NONE      ! Enforce explicit typing of all variables in this routine

      ! FUNCTION ARGUMENT DEFINITIONS:
      ! na

      ! FUNCTION PARAMETER DEFINITIONS:
      ! na

      ! INTERFACE BLOCK SPECIFICATIONS
      ! na

      ! DERIVED TYPE DEFINITIONS
      ! na

      ! FUNCTION LOCAL VARIABLE DECLARATIONS:
      ! na

RETURN

END FUNCTION <name>

```

Appendix F. Test File Documentation

Each test file should be documented (comments at the top of the file) following the guidelines below:

```
! <name of file>
! Basic file description: <specify number of zones, stories in building, etc>
! Highlights: <Purpose of this example file>
! Simulation Location/Run: <location information, design days, run periods>
! Building: <more details about building. metric units, if also english enclose in []{} or ()>
!
! Internal gains description: <lighting level, equipment, number of occupants, infiltration,
daylighting, etc>
!
! HVAC: <HVAC description and plant supply, as appropriate>
```

Several of the existing files are exemplary:

```
! 5ZoneAutoDXVAV.idf
! Basic file description: 1 story building divided into 4 exterior and
!   one interior conditioned zones and a return plenum.
! Building: single floor rectangular building 100 ft x 50 ft. 5 zones -
!           4 exterior, 1 interior, zone height 8 feet. Exterior zone
!           depth is 12 feet. There is a 2 foot high return plenum: the
!           overall building height is 10 feet. There are windows on
!           all 4 facades; the south and north facades have glass doors.
!           The south facing glass is shaded by overhangs. The walls
!           are woodshingle over plywood, R11 insulation, and gypboard.
!           The roof is a gravel built up roof with R-3 mineral board
!           insulation and plywood sheathing. The floor slab is 4 inches
!           of heavy concrete over 2 feet of dirt. The windows are double
!           pane 6mm clear with 6mm air gap. The window to wall ratio is
!           approximately 0.29.
!
!           The building is oriented 30 degrees east of north.
!
! Internal: lighting is 1.5 watts/ft2, office equip is 1.0 watts/ft2. There
!           is 1 occupant per 100 ft2 of floor area. The infiltration is
!           0.25 air changes per hour.
!
! HVAC: the system is a packaged VAV (DX cooling coil and gas heating coils).
!       the input is fully autosized.
```

```

! CoolingTower.idf
! Basic file description: This is a modification of the electric chiller test file
!                        with 2 single-speed cooling towers in series, UA = 175 for
!                        each tower.
!                        Shirey/Raustad, 2/9/01
!
!      Chiller Names      Type      Condenser      Capacity
!      Big Chiller        Electric   Water Cooled   100,000
!      Little Chiller     Electric   Water Cooled   20,000
!
! Run:      2 design days.
! Building: Fictional 3 zone building with interzone partitions connecting all zones together.
!           No ground contact (all floors are "partitions"). Roofs exposed to outdoor environment.
!           There is one single pane window.
! Internal: People, equipment, and lighting all at approximately "normal" levels and schedules.
! System:   3 zone terminal reheat system using a single air loop. Controlled about like the old
!           BLAST "NWS2" control profile. Heating up to 20 C during occupied hours, 15 C otherwise.
!           Cooling to 24 C while occupied, 30 C otherwise. Fans and coils scheduled to be unavailable
!           during unoccupied hours. Cooling coil off all winter. Reheat coil on all year.
!           System configuration is very basic--air loop has a fan and cooling coil, each zone leg has
!           nothing more than a reheat coil.
! Plant:    Heating loop served by purchased heating. Cooling loop served by two different types
!           of chillers and purchased cooling. Priority based controls determine which piece of equipment
!           tries to meet the load.
! SolDis=FullInteriorAndExterior, Aniso, Detailed Interior and Exterior Convection <== Should be
! in highlights.

```

And several are truly outstanding:

```

! 3-zone building with natural ventilation modeled by COMIS/EnergyPlus link. 3zvent.idf.
!
! See plan view of building, below, for location of air-flow openings and cracks.
!
! Natural ventilation takes places through openable exterior windows in WEST_ZONE and NORTH_ZONE
! and an openable interior door between WEST_ZONE and NORTH_ZONE. The ventilation is temperature
! controlled. Window1 in WEST_ZONE and DoorInSurface_3 (defined in WEST_ZONE) are opened when
! (1) the previous time step air temperature in WEST_ZONE is greater than the ventilation setpoint
! schedule value (a fixed value of 21.1C in this run) and (2) the previous time step air
! temperature in WEST_ZONE is greater than the outside air temperature.
! Similarly, Window2 in NORTH_ZONE is opened when (1) the previous time step air temperature
! in NORTH_ZONE is greater than the ventilation setpoint schedule value and the previous
! time step air temperature in NORTH_ZONE is greater than the outside air temperature.
!
! Controls to modulate the ventilation openings are applied to the WEST_ZONE but not the
! NORTH_ZONE (see the COMIS ZONE DATA inputs for these zones).
!
! The EAST_ZONE has no natural ventilation (Ventilation Control Mode = NoVent in the COMIS ZONE
! DATA input for this zone).
!

```

```
| In addition to natural ventilation in two of the zones, there are cracks in all exterior and
| interior walls. Cracks in the exterior walls allow infiltration of outside air into the zones.
| The interior doorway and cracks in the interior walls allow cross-mixing of air between zones
| (i.e., interzone air flow). There are no cracks in the roof surfaces.
```

```
| 
| * External Node NFacade
|                                     Window2      Crack
| *****[                          ]***//*****
| |
| | / Crack                                / Crack
| | /                                  NORTH_ZONE /
| | |
| * External   DoorInSurface_3          Crack
| Node         ****//[**] [*****/]**//*****/**//***** * External Node EFacade
| WFacade     | Crack                | Crack                    |
| |           |                     |                       |
| | / Crack    |                   / Crack                 / Crack
| | /          |                   /                        /
| |            WEST_ZONE             EAST_ZONE              |
| **//[*[       ]*****/]**//*****
|               Window1              Crack
|
| * External Node SFacade
```

```
| Plan view of 3-zone building showing location of openable
| windows ([ ]), openable door ([ ]) and cracks (/).
| Not to scale.
```



ENERGYPLUS™ QUESTIONNAIRE FOR CODE CONTRIBUTIONS

The EnergyPlus™ building energy simulation computer program has been developed jointly by the University of Illinois at Urbana-Champaign and Lawrence Berkeley National Laboratory (Berkeley Lab) under funding from the U.S. Department of Energy. Berkeley Lab has the sole authority to administer the licensing of EnergyPlus™ software.

To ensure the long-term viability of EnergyPlus, any proposed contributions must be made with “no strings attached” – that is, at a minimum, with royalty-free, non-exclusive, unlimited rights for Berkeley Lab to use, copy, modify, prepare derivative works, and distribute any contributions (both source code and executables), and to permit others to do so. Exceptions to this policy are made only in extraordinary circumstances, on a case-by-case basis, and only by Berkeley Lab’s Technology Transfer Dept.

This Questionnaire is intended to aid in our management of contributions to the EnergyPlus code base and to flag any intellectual property or licensing issues that may need to be resolved. EnergyPlus is a team effort! We appreciate your cooperation!

THIS FORM MUST BE FILLED OUT COMPLETELY FOR US TO CONSIDER YOUR CONTRIBUTION – THANKS!

Company/Institution (“Contributor”): _____

Name of responsible Contributor employee: _____

Title or position: _____

Department (if applicable): _____

Address: _____

City / State / Postal Code / Country: _____

Tel: _____ Fax: _____

E-Mail: _____ Web: http://_____

Who is your contact on the EnergyPlus Development Team? (or “None”) _____

☐ I have attached a brief description of my contribution (subroutine(s), module(s), library/ies, etc.). (THIS IS REQUIRED)

☐ Yes ☐ No ☐ Don’t Know -- Do you have an active E+ Collaborative Developer License Agreement in place?
If ‘yes’, is your contribution a user interface? ☐ Yes ☐ No

A. AUTHORSHIP

1. For the code you are submitting, did you or your fellow employees write every line of code? Before answering “yes,” you should actually contact your fellow employees to confirm that they did not use any code written by others.(e.g., “public domain code,” “open source code,” etc.).

☐ Yes ☐ No (If you don’t know, then find out.)

2. For the code you are submitting, was any written by a contractor or consultant?

☐ Yes ☐ No (If you don’t know, then find out.) ☐ Not applicable (i.e., I answered ‘yes’ to question #1)

2(a) Have you confirmed that the funding/contract document with such contractor/consultant grants you or your institution the necessary rights to provide a royalty-free unlimited license to your contributions to Lawrence Berkeley National Laboratory? (Note: if you are in an academic/research institution, you should confirm this with your contracts & grants office or your technology transfer office). If the answer is “No,” then such rights must be secured in writing before we can consider such code for incorporation into EnergyPlus.

☐ Yes ☐ No (If you don’t know, then find out.)

2(b) Did the contractors/consultants include any code that they did not *actually write themselves*? Before answering "yes," you should confirm with them that they did not use any code written by others (e.g., "public domain code," "open source code," etc.).

☐ Yes ☐ No (If you don't know, then find out.)

3. For ANY code that was not actually written by you, your fellow employees or a contractor/consultant, do you know the portions of the code written by others (i.e., the name of the subroutine, module, library, etc.)?

☐ Yes ☐ No ☐ Don't know ☐ Not applicable (no third party code included)

If "yes," please list all third party code here (if more than two pieces of third party code, attach separate sheets for each):

Name of Third Party Code #1: _____

Copyright notice

☐ None

☐ Printed out and attached

Written license agreement covering the code

☐ None

☐ Printed out and attached

If there is no written license agreement covering the code, then please attach on a separate sheet, any helpful background and contact information to aid in tracking down a proper written license agreement. Also, please note, that with rare exception, code that people consider to be "in the public domain" is almost never actually legally in the public domain.

Name of Third Party Code #2: _____

Copyright notice

☐ None

☐ Printed out and attached

Written license agreement covering the code

☐ None

☐ Printed out and attached

If there is no written license agreement covering the code, then please attach on a separate sheet, any helpful background and contact information to aid in tracking down a proper written license agreement.

B. FUNDING

1. For the code you are submitting, was your contribution funded under a Berkeley Lab R&D Subcontract?

☐ Yes ☐ No ☐ Don't know

2. For other funding sources, have you confirmed that the funding document (if any) grants you or your institution the necessary rights to provide a royalty-free unlimited license to your contributions to Lawrence Berkeley National Laboratory? (Note: for employees of academic or research institutions, you should confirm this with your contracts & grants office or your technology transfer office). If the answer is "No," then you do not have the necessary rights to such code and we cannot accept such code for consideration of incorporation into EnergyPlus.

☐ Yes ☐ No ☐ I funded this myself ☐ I don't know the funding source

To the best of my knowledge, all of the above is complete and correct. If there are any extenuating or exceptional circumstances regarding any of the above, I have attached a sheet to this form explaining same.

Signed: _____

Printed Name: _____

Date: _____

Please submit the completed **and signed** form via FAX or, if scanned, via e-mail to **BOTH**:

Linda Lawrie
Fax: (413) 294-9148
E-Mail: Linda@lawrie.com

Seth B. Rosen, Technology Transfer Dept., Berkeley Lab
Fax: 510-486-6457
E-Mail: SBRosen@LBL.gov

Thank you very much for your cooperation from the EnergyPlus Team!